UNIVERSITY OF CALIFORNIA,
IRVINE

Indexed and Distributed Processing of Similarity Selection and Join Queries.

DISSERTATION

submitted in partial satisfaction of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

in Information and Computer Science

by

Alexander Behm

Dissertation Committee:
Professor Chen Li, Chair
Professor Michael J. Carey
Professor Sharad Mehrotra

2013

DEDICATION

I dedicate this thesis to my loving wife and son, to my parents for their enduring moral and financial support, and to my brother for his infectious positivity and humor.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

I am deeply grateful to my advisers Professor Chen Li and Professor Michael Carey for guiding me through this work and shaping me into an independent researcher. Chen has patiently taught me how to develop, dissect, and present research ideas. He has ingrained in me a passion for creating high-quality software, and he has inspired me to accept difficult challenges. I thank Mike for sharing his wisdom and humorous anecdotes. Our discussions have enlightened me and often brightened my day.

I would like to thank Professor Sharad Mehrotra for joining my doctoral committee. His knowledge and enthusiasm have helped me to improve myself.

Vinayak Borkar deserves my thanks for his competent engineering leadership in the ASTERIX project and for his unwavering positivity.

I want to express my gratitude to Professor Volker Markl who sparked my interest in research while mentoring me at IBM. I would not be where I am today without him.

I would like to express my appreciation for the mentors who have enriched my academic pursuits with an industry perspective during internships. In particular, I thank Professor Peter J. Haas from IBM Research, Dr. Jingren Zhou and Dr. Ming-Chuan Wu from Microsoft Research, and Dr. Marcel Kornacker from Cloudera.

I thank my co-authors Professor Alin Deutsch, Athena Ahmadi, Hotham Altwaijry, Jarod Wen, Professor Jiaheng Lu, Lingjie Weng, Nagesh Honnalli, Dr. Nicola Onose, Pouria Pirzadeh, Raman Grover, Dr. Rares Vernica, Sattam Alsubaiee, Dr. Shengyue Ji, Professor Vassilis J. Tsotras, Professor Xiaohui Xie, Professor Yannis Papakonstantinou, Yasser Altowim, Yingyi Bu, Young-Seok Kim and Zachary Heilbron for our fruitful collaboration.

This research is partially supported by the NSF Awards #IIS-0238586, #IIS-0742960, #IIS-0844574, #IIS-1030002, the NSF funded RESCUE Project, the NIH grant 1R21LM010143-01A1, a Google research award and a Google Ph.D. fellowship award, gift funds from Microsoft and Yahoo, a research grant from Amazon, a fund from CalIt2, and a Ph.D. fellowship from the ARCS Foundation.

I thank IEEE for permission to include materials from "Space-Constrained Gram- Based Indexing for Efficient Approximate String Search" (ICDE 2009) and "Answering Approximate String Queries on Large Large Data Sets Using External Memory" (ICDE 2011) into my thesis. I also thank the Oxford University Press for allowing me to include "Hobbes: Optimized Gram-Based Methods for Efficient Read Alignment" (NAR 2012) into this thesis.

CURRICULUM VITAE

Alexander Behm

EDUCATION

Doctor of Philosophy in Information and Computer Science **2013**
University of California, Irvine *Irvine, California*

Bachelor of Science in Applied Computer Science **2006**
University of Cooperative Education *Stuttgart, Germany*

SELECTED HONORS AND AWARDS

Ph.D. Fellowship Award **2012**
Google Inc.

Ph.D. Fellowship Award **2010-2012**
National Achievement Rewards for College Scientists (ARCS)

PUBLICATIONS

ASTERIX: An Open Source System for "Big Data" Management and Analysis (Demo) **2012**
International Conference on Very Large Databases (VLDB)

ASTERIX: Scalable Warehouse-Style Web Data Integration **2012**
International Workshop on Information Integration on the Web (IIWeb)

Hobbes: Optimized Gram-Based Methods for Efficient Read Alignment **2011**
Oxford Journals Nucleic Acids Research (NAR)

ASTERIX: Towards a Scalable, Semistructured Data Platform for Evolving World Models **2011**
Journal of Distributed and Parallel Databases

Answering Approximate String Queries on Large Large Data Sets Using External Memory **2011**
IEEE International Conference on Data Engineering (ICDE)

Supporting Location-Based Approximate-Keyword Queries **2010**
ACM International Conference on Advances in Geographic Information Systems (GIS)

Space-Constrained Gram-Based Indexing for Efficient Approximate String Search **2009**
IEEE International Conference on Data Engineering (ICDE)

Integrating Query-Feedback Based Statistics into Informix Dynamic Server **2007**
GI-Fachtagung Datenbanksysteme für Business, Technologie und Web (BTW)

ABSTRACT OF THE DISSERTATION

Indexed and Distributed Processing of Similarity Selection and Join Queries.

By

Alexander Behm

Doctor of Philosophy in Information and Computer Science

University of California, Irvine, 2013

Professor Chen Li, Chair

A similarity query is to find from a collection of items those that are similar to a given query item. Answering similarity queries is important in applications such as DNA sequence assembly, record linkage, and query relaxation, where errors could occur in queries as well as the data. The wide relevance of similarity queries presents to us application-specific constraints and desiderata. In this thesis, we develop and evaluate indexes and algorithms for answering such queries efficiently in the context of four different settings. First, we present a DNA-specific alignment method whose primary focus is on the speed of query execution. Our results show a performance improvement of 2-10X compared to existing state-of-the-art packages. Second, we develop a flexible compression technique for reducing the size of an inverted index to a given amount of space while retaining efficient query processing. In our experimental evaluation we could reduce the index size up to 60% without sacrificing query response times. Third, we study external-memory solutions well-suited for database management systems, where the data and indexes are stored on disk, and demonstrate a substantial benefit over alternative methods. Fourth, we discuss how we incorporated some of our solutions into the ASTERIX parallel database system to optimize similarity queries via secondary indexes. We elaborate on our design choices and query-processing capabilities, and conclude with experiments on large-scale data.

Chapter 1

Introduction

Many applications require answering similarity queries. Given a collection of items, a similarity selection query asks for all items in the collection that are similar to a given query item. Analogously, a similarity join query finds all similar pairs of items from two collections. Such querying capabilities are useful in a wide variety of applications, most notably (but not limited to) those that deal with textual data. For example, in data cleaning [86] we wish to discover and eliminate inconsistent, incorrect, or duplicate entries of the same real-world entity such `Arnold Schwarzenegger` vs. `Arnold Schwarseneger` or `Wall-Mart` vs. `Wal Mart`. Similar problems exist in record linkage [97, 118, 119] where we aim to fuse records from different data sources that belong to the same entity. Matching such entities is often a challenging task due to errors and inconsistencies in the different data sources. Not only can the data contain real errors but also slightly different yet valid representations of the same thing such as `P.O. Box` vs. `PO Box` or `Sunshine Ave.` vs. `Sunshine Avenue`. Another interesting application, query relaxation, is illustrated by Google's famous listing of the countless misspellings of `Britney Spears`[1]. The goal of query relaxation/suggestion systems is to automatically refine or suggest a more relevant query to the user. The `Britney Spears`

[1] http://www.google.com/jobs/britney.html

example is illustrative of a user's potential frustration since there are many valid spellings of the name `Britney`, but only one of them refers to the popular singer. Other applications of similarity queries include entity extraction [116] and iris code matching [50]. So far, our examples have focused on the special class of string similarity queries. Another important class is that of set-similarity queries, where we wish to find similar sets of items. Among others, set-similarity queries are useful in applications such as information retrieval, near duplicate detection [51], fraud detection [52, 91], and recommendation mining [102]. For example, a social networking site could mine for friend recommendations by identifying users with similar interest sets, such as a user with interests {`Cooking, Tennis, Music, Reading, Politics`} and another one with interests {`Cooking, Programming, Music, Reading`}.

Resolving data quality issues is becoming increasingly important as businesses and governments warehouse their data for analysis purposes. Data-driven decision making follows the "garbage in, garbage out" principle, meaning that those decisions are only as good as the data they are based on. For example, one can imagine over- or underestimating the number of employees in different departments of a business, or the corresponding aggregate expenses on salary, etc. Further, even a small number of errors can potentially have large impacts on certain types of analyses [9]. Data quality is a serious financial concern. The Data Warehousing Institute [37] estimated that $611 billion are lost annually in the US due to poor customer-data quality. The report also states that as many as 2% of the records in customer databases become obsolete within one month due to deaths, name changes, etc. The importance of data quality is also reflected in recent market estimates. For example, in 2012 Gartner [6] had estimated that revenue in Master Data Management software would reach $1.9 billion that year, and that it would climb to $3.2 billion by 2015. The market for data-quality tools specifically was estimated to be $950 million in 2011 [41] with an annual expected growth rate of 14%.

Answering similarity queries efficiently is challenging for a variety of reasons. First, many

widely-applicable similarity functions such as edit distance and Jaccard are computationally expensive, and therefore, brute-force approaches to answering similarity selection and join queries are impractical even for small data sets. Second, an increasing number of applications need to deal with large amounts of data, and therefore, developing efficient and scalable solutions is especially important in today's Web and "Bid Data" era. Third, due to the wide applicability of similarity queries it stands to reason that a single solution cannot meet the needs of all possible applications. In particular, for such a diverse set of applications we expect different data characteristics and sizes, similarity functions, hardware environments, query workloads, and performance requirements.

In this thesis, we have develop and evaluate several index-based methods for answering similarity queries. Our focus is on designing efficient solutions for answering queries based on commonly used similarity functions such as edit distance and Jaccard. We have identify potential issues that applications might face, and address them with specialized techniques summarized in the following preview of our contributions.

In Chapter 3 we will serve the important use case of DNA sequencing [33, 70], which is an indispensable tool in many areas of biology and medicine. Modern high-throughput sequencing machines can produce large amounts of DNA snippets quickly and cheaply. Mapping such DNA snippets to a known reference sequence often represents the first step in the computational analysis of sequencing data. Due to sequencing errors and/or genetic variations, many DNA snippets may map to a target reference sequence approximately but not exactly, and therefore, we require string-similarity querying capabilities. We will present a gram-based read-mapping package called Hobbes. Hobbes implements two novel techniques that yield substantial performance gains: an optimized gram-selection procedure, and a cache-efficient filter for pruning candidate mappings. We will show that Hobbes outperforms several state-of-the-art read-mapping programs, including Bowtie, BWA, mrsFast, and RazerS.

In another setting, some applications may require very low response times for similarity

queries on large data, and therefore it is desirable to hold the supporting indexes in memory. To make more efficient use of available hardware resources it is important to reduce the memory footprint of the indexes supporting similarity queries. In Chapter 4 we will present two techniques for reducing the size of a q-gram inverted index for answering global-alignment similarity queries. Such q-gram inverted indexes are known for their large size relative to the data they index. We will present two independent approaches for compressing a q-gram inverted index to a given amount of space while retaining efficient query processing. The first approach is based on the idea of discarding some of the inverted lists. The second approach minimizes redundancy in the index by combining some of the correlated inverted lists. We will discuss the effects of such compression on queries, and develop cost-based algorithms for constructing an index based on a given memory constraint and a given query workload. We will conclude with experiments showing that we could reduce index sizes by up to 60% without sacrificing query response times.

In Chapter 5 we will study external-memory solutions for answering global-alignment similarity queries. Our approach is well-suited for scenarios where indexes and data are assumed to be and disk, e.g., in database management systems. We will propose a multi-level inverted index to minimize the I/O costs of answering queries. We will present a new storage layout for an inverted index that is optimized for answering similarity queries, and we will show how to efficiently construct such an index with limited buffer space. The new layout is based on partitioning the data items such that at query time only a few partitions must be considered, significantly reducing the number of false positives. To answer queries efficiently, we will develop a cost-based algorithm that balances the I/O costs of accessing inverted lists and candidate answers. Our experimental results show that the combination of our techniques outperforms a standard inverted index as well as a recently proposed tree-based index, called BED-Tree.

We will discuss how we incorporated similarity querying capabilities into the ASTERIX

parallel database management system in Chapter 6. ASTERIX is a shared-nothing database system for storing, indexing, and querying large quantities of semi-structured data. It is geared towards the new breed of "Big Data" applications which aim to derive knowledge from the vast quantities of data being generated on a daily basis on the web, for example, in blogs, social networks, online communities, click streams, etc. Data gathered from said sources is bound to contain noise and inconsistencies, since many users publish their data informally resulting in abbreviated keywords and misspellings. We will present how to express set-similarity queries in the declarative ASTERIX query language, and introduce data-definition language constructs to create secondary indexes for optimizing similarity queries. To use secondary indexes in query processing, we will develop the corresponding rewrite rules used in the ASTERIX optimizer. We will also detail the implementation of our secondary indexes supporting similarity queries. We will experiment with different query-plan alternatives on large-scale data for similarity selection and join queries, and compare our index-based join approach with an existing unindexed approach.

Chapter 2

Foundations and Related Work

In this chapter we introduce fundamental concepts, similarity measures, and existing techniques for answering similarity queries. For simplicity, we focus on answering string-similarity queries, but many of the techniques also apply to answering set-similarity queries, as will become clear shortly. We also touch upon the large body of existing work in this area, and leave room for more detailed discussions in the respective chapters.

2.1 Similarity Selection and Join Queries

Similarity Selection Queries: A similarity *selection* or *range* query consist of a string r and a similarity threshold k. Given a set of strings S, the query asks for all strings in S whose similarity to r is greater than or equal to k (or no less than k for distance measures).

Similarity Join Queries: Analogously, given two string collections R and S, and a similarity threshold k, a similarity join query asks for all pairs $\langle r, s \rangle$ of strings in R and S whose similarity is greater or equal to k (or no less than k for distance measures).

We use the terms "string similarity query" and "approximate string query" synonymously.

2.2 Tokenizing Strings into Q-Grams

Many approaches to answering string-similarity queries rely on q-grams. The q-grams of a string s are all its substrings of length q, obtained by sliding a window of length q over s. For instance, the 2-grams of a string `cathey` are $\{\texttt{ca}, \texttt{at}, \texttt{th}, \texttt{he}, \texttt{ey}\}$. For a string s we use "$|s|$" to denote its length, and $G(s)$ to denote its multiset of q-grams. The number of q-grams in $G(s)$ is $|G(s)| = |s| - q + 1$.

Prefixing and Suffixing: For better filtering performance, it is sometimes useful to artificially extend a string's set of q-grams. We introduce two special characters α and β. Given a string s and a positive integer q, we extend s to a new string s' by prefixing $q-1$ copies of α and suffixing $q-1$ copies of β. The new string s' generates $|s| + q - 1$ q-grams. For instance, suppose $\alpha = \#$, $\beta = \$$, $q = 2$, then prefixing and suffixing the string `cathey` to $\#\texttt{cathey}\$$ results in this set of 2-grams $\{\#\texttt{c}, \texttt{ca}, \texttt{at}, \texttt{th}, \texttt{he}, \texttt{ey}, \texttt{y}\$\}$.

For ease of exposition, we sometimes omit the prefixing and suffixing step. Our techniques do not rely on prefixing and suffixing, though in practice, it is often beneficial to performance.

2.3 Similarity and Distance Measures

Various measures can quantify the similarity between two strings. We focus on the following similarity/distance metrics, and often use the term "similarity" to loosely refer to similarity or distance functions.

Hamming distance: Given two strings r and s of equal length, the hamming distance is

the number of single-character substitutions required to transform one string into the other. For example, the hamming distance between "`kate`" and "`caty`" is 2.

Edit distance: The edit distance [75], a.k.a. Levenshtein distance, between two strings r and s is the minimum number of single-character insertions, deletions, and substitutions needed to transform r to s. For example, the edit distance between "`cathey`" and "`kathy`" is 2, because "`cathey`" can be transformed into "`kathy`" by substituting "`c`" for "`k`" and deleting "`e`". A classic algorithm for computing the edit distance between two strings uses dynamic programming based on the following Equation 2.1

$$M(i,j) = \begin{cases} r[i] = s[j] : M(i-1, j-1) \\ r[i] \neq s[j] : 1 + \min \begin{cases} M(i-1,j) \\ M(i,j-1) \\ M(i-1,j-1) \end{cases} \end{cases} \tag{2.1}$$

We create a matrix $M(i,j)$ where each cell represents the edit distance between the substrings of s and r from their first character up to character i and j, respectively. To illustrate the resulting algorithm, Figure 2.1 shows an example of a populated dynamic programming matrix to derive the edit distance between "`cathey`" and "`kathy`". We initialize the first row to contain the edit distance between the empty string and s = "`kathy`"'s substrings from character 1 up to $|s|$. The first column is populated analogously for r = "`cathey`". We then compute the remaining cells in a row-wise or column-wise fashion by applying the recurrence function from Equation 2.1. The final edit distance is found in the last cell, as highlighted in the figure.

Set-similarity measures: Another class of measures, the set-similarity functions, focuses on expressing the similarity between two sets. We may utilize such set-similarity functions to quantify the similarity between two strings by converting the strings into multisets. We

		k	a	t	h	y
	0	1	2	3	4	5
c	1	1	2	3	4	5
a	2	2	1	2	3	4
t	3	3	2	1	2	3
h	4	4	3	2	1	2
e	5	5	4	3	2	2
y	6	6	5	4	3	**2**

Figure 2.1: Computing the edit distance between "`cathey`" and "`kathy`".

could tokenize the strings into q-grams or words, and declare their string similarity to be the set similarity of their token multisets. For example, a common set-similarity measure is Jaccard. Using our notion of q-grams, the Jaccard similarity is given by:

$$Jaccard(G(r), G(s)) = \frac{|G(r) \cap G(s)|}{|G(r) \cup G(s)|}. \tag{2.2}$$

2.4 Query Answering with Q-Gram Inverted Indexes

Answering string-similarity queries using q-grams is based on the following observations made in [108]. A string r that is within hamming/edit distance k of another string s must share at least the following number of grams with s:

$$T = \max(|G(r)|, |G(s)|) - k \times q. \tag{2.3}$$

Intuitively, every hamming/edit operation can affect at most q grams of a string, and T is the minimum number of grams that "survive" k edit operations, which could destroy at most $k \times q$ grams. Equation 2.3 reflects that we may compute the lower bound T from the perspective of r or s and take the maximum. In the remainder of this thesis, we compute

the T lower bound from the query r's perspective, simplifying the equation to:

$$T = |G(r)| - k \times q. \tag{2.4}$$

Lower bounding set-similary functions: Given a threshold on the similarity, such a T lower bound on the number of grams (tokens) can also be derived for other common similarity functions such as Jaccard [76]. Most of the techniques discussed in this thesis (except Chapter 3) are applicable to those functions as well, and our solutions are not limited to string-similarity queries, but also apply equally to set-similarity queries using sets of arbitrary elements (not necessarily q-grams).

Next, we will show how to use the T lower bound to efficiently answer string-similarity queries based on q-gram inverted indexes. For clarity, we distinguish between two types of related string-similarity problems: the global and local alignment problems [96], explained in the following Subsections 2.4.1 and 2.4.2.

Our contributions in Chapter 3 focus on the local alignment, whereas the rest of this thesis (Chapters 4-6) deals with global aligmnent.

2.4.1 Global Alignment: Finding Similar Strings

In the *global aligmnent* [96] version of approximate string matching we wish to find from a collection of strings those similar to a given query string. For answering such queries we rely on q-gram inverted indexes. For each gram g in the strings in S, we have an inverted l_g of the ids of the strings that include this gram g. For instance, Figure 2.2 shows a collection of strings and the corresponding inverted lists of their 2-grams. To construct the index, we decompose each string into its grams and add its string id to the inverted lists of its grams.

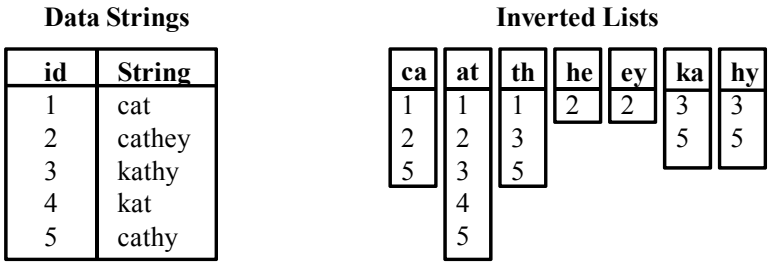


Figure 2.2: A string collection of person names and their inverted lists of 2-grams.

Answering string similarity queries can be approached by solving a "T-Occurrence Problem" [104], also known as q-gram counting, defined as follows:

> *T-Occurrence Problem*: Find the string ids that appear at least T times on the inverted lists of the query's grams, where T is a constant related to the similarity function, the threshold in the query, and the gram length q.

T is the lower bound on the number of grams any answer must share with r, as introduced in Equation 2.4. To answer a string-similarity query, we decompose the query string into its grams and retrieve their corresponding inverted lists. We process those inverted lists to find all string ids that occur at T times. Solving the T-occurrence problem yields a set of candidate string ids, which are a superset of the true answers to the query. To remove false positives, we fetch the strings corresponding to the candidate string ids and compute their real similarities to the query in a post-processing step.

Note that if the threshold $T \leq 0$, then the entire data collection needs to be scanned to compute the results. We call it a *panic case*.

2.4.2 Local Alignment: Finding Similar Substrings

The *local aligmnent* [96] problem refers to finding all substrings in a long reference sequence that are similar to a given query pattern. Finding substrings in a reference sequence that

share at least T q-grams with a given query string can be facilitated with an inverted index on q-gram positions, explained as follows for Hamming distance. Figure 2.3 shows an example of a 5-gram inverted index created on a long DNA reference sequence.

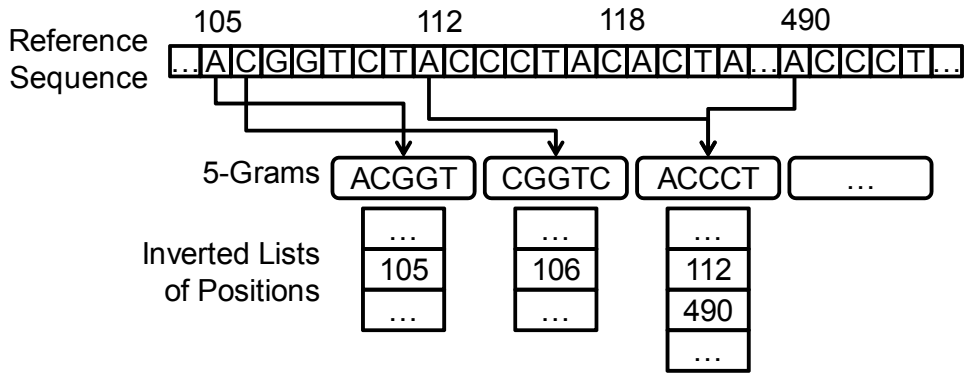


Figure 2.3: Excerpt of a DNA reference sequence and its 5-gram inverted index. The inverted lists of the 5-grams `ACGGT`, `CGGTC`, and `ACCCT` are shown, each containing a sorted list of positions in the reference sequence at which the respective 5-gram appears.

To find all positions similar to the query string ACGGTCTTCCCTACGGT within Hamming distance $k = 2$ and $T = 17 - 5 + 1 - 2 * 5 = 3$, we first look up the query's 5-grams in the inverted index. Notice that only the grams `ACGGT` and `CGGTC` (underlined in the query) are present in the index. We traverse their inverted lists, and normalize each element relative to the position of the corresponding gram in the query. For example, the 5-gram `CGGTC` appears at position 2 in the query string, so the relative position of the element on `CGGTC`'s inverted list is $106 - 2 + 1 = 105$. In this way, we can count how many of the query's grams are contained in substrings of the reference sequence starting at a fixed position (position 105, in this example). The gram `ACGGT` appears twice in the query, and we treat each occurrence as a separate inverted list. Its appearance at position 1 yields a normalized list of $\{105 - 1 + 1 = 105, 118 - 1 + 1 = 118\}$, and a list $\{105 - 14 + 1 = 92, 118 - 14 + 1 = 105\}$ for position 14. Next, we count the occurrences of each element on the normalized lists. The positions 92 and 118 are pruned according to the q-gram counting filter, because their occurrences do not meet the lower bound of $T = 3$. Position 105 has a count of 3, hence it is a candidate answer whose Hamming distance to the query still needs to be computed.

2.5 Related Work

In this section, we touch upon existing work relevant to string-similarity query processing. We direct the reader to the last paragraph in this section for a list of comprehensive surveys of the field. We also discuss related work in more detail separately in each chapter.

There are two main families of algorithms for answering approximate string queries: those based on trees (e.g., suffix trees, tries), and those based on inverted indexes of q-grams.

Tree Methods: The tree-based approaches [90, 55, 124] mostly focus on the local alignment problem based on edit distance and similar functions, though they can also support global alignment. To allow mismatches during the index traversal, such tree-based methods employ backtracking [71], automatons [63, 59, 29], or dynamic programming [90].

Inverted Index Methods: The gram-based methods (supported by an inverted index) [47, 76, 18] follow a filter-and-verify paradigm. Using gram counting (Section2.4.1), they first identify a set of candidate answers, and then verify the true distance for those candidates to remove false positives. Apart from those solutions that use "regular" overlapping q-grams [47, 76], other variants of filtering use multiple patterns based on the pigeonhole principle [106], non-overlapping grams [99], gapped grams [15, 23], or variable-length grams [77, 123].

Similarity Join Queries: Many algorithms have been developed for the problem of approximate string joins based on various similarity functions [13, 16, 104, 120, 121, 114, 78, 115, 122], especially for record linkage and/or data cleaning [68]. Some of them have been proposed in the context of relational DBMS systems [44, 26, 28]. Most solutions rely a form of q-gram filtering, but some approached the problem with tries as well [113]. There is also recent work on parallel processing of similarity joins using the MapReduce framework [110, 112].

Similarity Selection Queries: Several recent papers focused on approximate selection

queries [47, 76, 111, 124, 21]. Hore et al. [53] proposed a gram-selection technique for indexing text data under space constraints, mainly considering SQL LIKE queries. There are recent studies on the problem of estimating the selectivity of SQL LIKE substring queries [27, 57, 69], and approximate string queries [87, 61, 72, 49]. Other related studies include [40, 65, 103]. There has also been a lot of work in the bio-informatics community for approximately mapping DNA or protein snippets to a known reference sequence [39, 71, 80, 83].

Relevant Surveys: Excellent, comprehensive surveys of the field have been conducted in [96], and more recently [48]. Another survey discusses the state-of-the-art in approximate string matching for the important application of DNA sequence alignment [81]. Overviews of data cleaning and record linkage are presented in [119, 86]. A recent survey on inverted indexes for information retrieval [126] is also relevant to our q-gram based approach.

Chapter 3

Hobbes: DNA Sequence Mapping

3.1 Introduction

DNA sequencing is an indispensable tool in many areas of biology and medicine. Recent advances in high-throughput sequencing have made it possible to sequence billions of bases quickly and cheaply. These technological breakthroughs have opened the door for personal genome sequencing and for the creation of a number of new tools for studying diseases, genomes, and epigenomes. Such new sequencing methods have improved the daily sequencing throughput by a factor of up to 1000 [66], and decreased the cost to a fraction (4-0.1%) of that of older technologies such as Sanger sequencing. While the original Human Genome Project cost $3 billion over 10 years, today it is possible to decode a whole human genome for $1000 in a single day [32]. This dramatic improvement has revolutionized the field of genomics, and enabled even single research groups to experiment with large amounts of cheaply produced sequence data. For example, the HiSeq platform from Illumina can produce 6 billion 100bp (100 base pairs) reads within only 11 days. The SOLiD system from Life Technologies can generate over 20GB of DNA sequences per day. As a result, this new

onslaught of sequencing data poses significant challenges to the computational infrastructure for storing and analyzing it. The dominating cost of many applications has shifted towards the processing associated with computationally analyzing genomic data [89].

Mapping DNA snippets (reads) from high-throughput sequencers to a reference sequence often represents the first step in the computational analysis of sequencing data in many applications. Figure 3.1 illustrates this read-mapping process. First, a high-throughput sequencing machine generates a (typically large) set of reads. Next, we map those reads to a known reference sequence to determine their matching locations. Those mapping locations and their alignment information (e.g. whether there are mismatches) are used in application-specific downstream analyses.

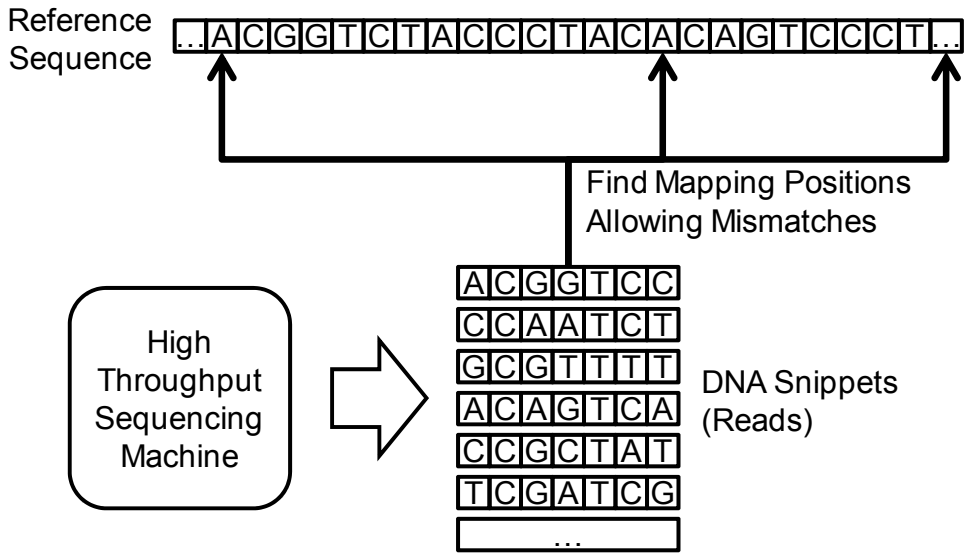


Figure 3.1: An illustration of the read-mapping problem.

The enormous amount of reads produced from the sequencers poses a great challenge on the speed and the accuracy of read alignment programs for two major reasons. First, the reference sequence can be very large. For instance, the human genome is about 3 billion base pairs (bp) long. Mapping a billion reads to the human genome volume to checking 3×10^{18}

candidate locations. Second, due to sequencing errors and/or genetic variations, many reads map to the reference sequence approximately but not exactly, and therefore, to map a read to the reference sequence, read mapping programs should allow a certain number of mismatches between the read and a candidate location. Although a number of high-performance read alignment programs have been developed, it still takes days or even weeks (depending on different mapping criteria) to align billions of reads to a large reference genome on a single desktop. As the sequencing technology is progressing toward generating longer reads, and thus requiring read-mapping programs to be able to handle more mismatches, the need for faster and more accurate read alignment programs is greater than ever.

Related work: Existing approaches to the read-mapping problem can be broadly classified into two categories: trie-based methods and gram-based methods. In the first group, most of the popular packages use the Borrows-Wheeler Transform (BWT) [24] and usually FM index [39] to encode their trie, e.g., Bowtie [71], BWA [80], and SOAP2 [83]. These packages have a small memory footprint (about 2 GB), and are efficient for finding a few mappings for short reads without too many mismatches. They use backtracking to allow mismatches during the tree traversal, and therefore, their performance deteriorates as the read length and the number of mismatches increase. BWT-based packages are typically not designed for finding a large number of mappings per read.

The second group, the gram-based methods, follow a filter-and-verify paradigm. Using grams, they first identify a set of candidate mappings, and then verify the true distance for those candidates to remove false positives. The candidate-generation step is often supported by an inverted index on grams (from the reads and/or the reference sequence), leading to a relatively large memory footprint. Early packages such as SSAHA [99] and BLAST [11] had long mapping times, infeasible for large data sets. Newer packages such as Maq [82], RMAP [107], ZOOM [85], SHRiMP [101], RazerS [117], mrsFAST [46], and mrFAST-CO [10] offer improvements, but they do not consistently outperform BWT-based methods. We will

show that our approach, Hobbes, outperforms both existing gram-based and BWT-based methods. A recent gram-based method called WHAM [84] has been developed in parallel to Hobbes. WHAM also proposes significant improvements to the gram-based approach and many of its techniques could be combined with those implemented in Hobbes.

Hobbes: Applications may differ in their requirements on a read-mapping package. Sometimes finding a couple of mappings per read is sufficient, but other times the application may need all mapping positions. For instance, in RNA-seq applications, due to the occurrence of homologous genes and multiple RNA isoforms[1] originating from the same gene, finding all mapping positions will be necessary for quantifying the expression level of a particular gene isoform [60]. Similarly, in ChIP-seq applications, finding all mapping positions is a necessary step for characterizing protein binding patterns in repeat regions of a genome [98, 30].

In this chapter we introduce Hobbes, a new gram-based local alignment method for short reads supporting Hamming and edit distance. Hobbes implements two novel techniques that yield substantial performance improvements: an optimized gram-selection procedure for reads, and a cache-efficient filter for pruning candidate mappings. We have systematically tested the performance of Hobbes on both real and simulated data with read lengths varying from 35 to 100bp, and compared its performance with several state-of-the-art read-mapping programs, including Bowtie, BWA, mrsFast, and RazerS. Hobbes is faster than all these read mapping programs while maintaining high mapping quality. In particular, Hobbes is about 2-10 times faster than state-of-the-art packages when finding all mappings per read, and it performs comparably when looking for a few mappings. Hobbes is also at least as accurate as the other packages. Hobbes supports the SAM output format and is publicly available at http://hobbes.ics.uci.edu.

In the following sections, we first identify two performance bottlenecks of existing gram-based approaches and then make two major contributions to overcome them. First, we present

[1] An RNA or gene isoform refers to a particular transcription of a DNA subsequence into mRNA.

a novel technique for judiciously choosing a small set of grams of each read to generate candidate mappings. Second, we develop a cache- and CPU-efficient filter for removing false positive mappings during the traversal of inverted lists.

3.2 Index Construction and Mapping Procedure

We first discuss how to map reads to the reference sequence with a given Hamming-distance threshold, and later extend our solution to support edit distance. As discussed in Section 2.4.2, our approach is based on generating overlapping q-grams of the reference sequence, and constructing an inverted index of those q-gram positions. To map a read, we generate its q-grams, and access the inverted index to compute a superset of all mapping positions. We then remove false-positive positions by computing the real distance of the read to the subsequences starting at those positions in the reference sequence.

3.2.1 Performance Issues of the Basic Q-Gram Method

For a long reference sequence, the filter-and-verify approach approach for mapping reads could suffer from the following performance problems:

1. *CPU-intensive gram counting.* Using all of a read's relevant inverted lists for gram counting can be expensive if there are many of them, or if some of the lists are very long. The cost of gram counting is directly related to the total number of elements on those inverted lists.

2. *Cache misses during candidate verification.* CPU caches are very small but fast memories that act as intermediaries to main memory. Transferring new data into the cache (a "cache miss") can last hundreds of CPU cycles. Accessing random portions of a very

long sequence has low locality, and hence, it can become a performance bottleneck due to cache misses.

In the following sections we present a new technique to judiciously select a few grams from a read to overcome the first performance issue. To tackle the second issue, we develop a novel filter based on augmenting the inverted lists with additional information.

3.2.2 Judiciously Selecting Q-Grams From Reads

In this section we aim to reduce the cost of processing inverted lists to generate candidate mappings. We present a new technique to judiciously select a few grams from a read to minimize the number of corresponding inverted lists, and the number of inverted-list elements that need to be considered for mapping the read.

3.2.2.1 Existing methods for q-gram selection

First, we briefly summarize the main ideas already developed in the literature to reduce the number of grams.

Prefix Filter: Consider a read s with a gram set $G(s)$ and the lower bound T as in Equation 2.4. Let the *prefix gram set* of read s be the $|G(s)| - T + 1$ least frequent grams in $G(s)$, i.e., with the shortest inverted lists. A candidate mapping must share at least one gram with the prefix gram set of s, because otherwise it could only reach a maximum count of $T - 1$ [28].

Shortened Prefix: Xiao et al. [120] use the positions of q-grams to reduce the size of the prefix gram set in the context of edit-distance-based joins. Their solution imposes a global ordering on the grams based on their frequency to achieve a consistent notion of prefix gram

set across all strings, and constructs a q-gram inverted index on prefix grams on-the-fly. To improve the index-construction time (the dominating factor), they reduce the prefix gram set of each string to the first $i \leq |G(s)| - T + 1$ grams in the global ordering, which contain $d + 1$ non-overlapping grams, where d is a given edit-distance threshold. Inspired by their work, we develop a new method to judiciously select a few q-grams for probing our index.

3.2.2.2 Optimal q-gram prefix selection

Recall that a substitution can affect at most q-grams (Equation 2.4). The insight that these q affected grams must be overlapping leads us to develop the following lower bound based on the positions of q-grams.

Lemma 1. *(Position-Based Prefix) Given a sequence s and its q-gram set $G(s)$, let P be a subset of $d + 1$ non-overlapping q-grams from $G(s)$. Then each sequence within Hamming distance d of s must have a gram in P.*

The intuition of the lemma is as follows. A substitution at position p can affect at most q overlapping grams, namely those starting from a position in $[p - q - 1, p]$. Since non-overlapping grams are at least q positions apart from each other, d substitutions can affect at most d non-overlapping grams. Among $d + 1$ non-overlapping grams, at least one gram remains unaffected by d substitutions. Since this analysis is true for every subset P of $d + 1$ non-overlapping grams of $G(s)$, we can generalize the lower bound as follows.

Lemma 2. *(Generalized Position-Based Prefix) A sequence r within Hamming distance d of sequence s must share at least k grams with every subset of $d + k$ non-overlapping q-grams of s.*

Optimal prefix selection: We want to select a set of prefix grams that is optimal in the sense that (1) it refers to a minimum number of inverted lists, and (2) those inverted lists

have the minimum total number of elements. The position-based prefix described above satisfies (1), but a read could have many possible sets of $d+1$ non-overlapping q-grams. To satisfy (2) we develop the following dynamic programming algorithm to select that set of $d+1$ non-overlapping q-grams from the read, which minimizes the total number of corresponding inverted-list elements.

Subproblem: Let $1 \leq i \leq d+1$ and $1 \leq j \leq |G(s)| - d*q$ be two integers. Let $M(i,j)$ be a lower bound on the sum of the lengths of the inverted lists of i non-overlapping grams starting from a position no greater than $j+(i-1)*q$. Our goal is to compute $M(d+1, |G(s)|-d*q)$.

Initialization: Let $L[p]$ denote the inverted list corresponding to the q-gram at position p, and $L[p].len$ its length. We initialize the row $M(0,j)$ to zero, and the column $M(i,0)$ to infinity.

Recurrence function:

$$M(i,j) = \min \begin{cases} M(i,j-1) \\ M(i-1,j) + L[j+(i-1)*q].len \end{cases} \tag{3.1}$$

Example: Figure 3.2 shows an example of finding an optimal q-gram prefix of a read $\mathtt{s} = \mathtt{GGTCTCACCCTGAACTAA}$, gram length $q = 5$, and Hamming distance threshold $d = 2$. An optimal set of positions of the $d+1 = 3$ non-overlapping q-grams are highlighted with a circle, a diamond, and a pentagon, including the cell in the matrix from which the q-gram position can be inferred. For example, "4" is the minimum value in the first row, and since its first appearance is in column 2, we can infer that the first optimal q-gram position is at $2 + (1-1)*q = 2$.

Pseudo Code: Figure 3.3 shows the pseudo code for finding an optimal set of prefix grams using dynamic programming. First, we create and initialize a matrix with r rows and c

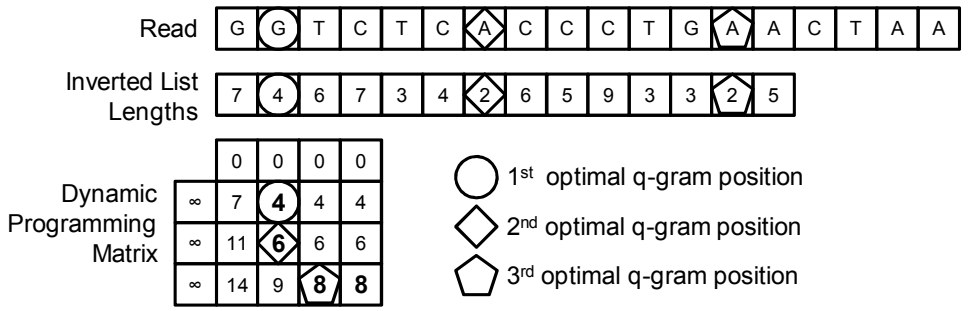


Figure 3.2: Example of dynamic programming algorithm for finding prefix grams with a read `GGTCTCACCCTGAACTAA`, gram length $q = 5$, and Hamming distance $d = 2$. Optimal gram positions are highlighted with a circle, a diamond, and a pentagon.

columns (lines 1-10). The matrix has $r = d + 1 + 1$ rows because we are looking for $d + 1$ non-overlapping grams and we create one extra row to avoid accessing an element out of bounds when applying the recurrence function. The number of columns is determined by subtracting $d * q$ from the number of inverted lists $|L|$ because we need at least $d * q + 1$ inverted lists to cover $d + 1$ non-overlapping grams. Again, we allocate one extra column to not go out of bounds. Next, we populate the matrix row-wise by applying the recurrence function (lines 12-22). As we find better gram positions resulting in a reduction of total inverted-list elements, we update the list of optimal gram positions to return (lines 15-17).

Complexity: The complexity of the algorithm for finding an optimal prefix for a read s with length $|s|$ and Hamming distance d is $O(|s| * d)$. Notice that the cost of the algorithm decreases when we increase d and q, because there are fewer sets of non-overlapping grams to choose from. For example, in Figure 3.2, we need to populate 12 matrix cells for a read of length 18. Our experiments show that the algorithm performs very well for well-chosen d and q values in real data sets.

```
Input:
    Gram length q.
    Inverted lists L corresponding to a read s's q-grams G(s).
    Hamming-distance threshold d.
Output:
    A set P of d + 1 non-overlapping q-gram positions,
    minimizing the sum of their inverted-list lengths.
Method: FindOptimalPrefix
1.  r = d + 1 + 1; // Number of rows.
2.  c = |L| − d ∗ q + 1; // Number of columns.
3.  Create matrix M(r, c);
4.  // Initialize matrix.
5.  FOR i = 1 TO c
6.      M(i, 0) = ∞;
7.  END FOR
8.  FOR j = 1 TO r
9.      M(0, j) = 0;
10. END FOR
11. // Populate dynamic programming matrix.
12. FOR i = 1 TO c
13.     FOR j = 1 TO r
14.         cmp = M(i − 1, j) + L[j + (i − 1) ∗ q].len;
15.         IF cmp < M(i, j − 1) THEN
16.             M(i, j) = cmp;
17.             P[i] = j + (i − 1) ∗ q;
18.         ELSE
19.             M(i, j) = M(i, j − 1);
20.         ENDIF
21.     END FOR
22. END FOR
23. RETURN P;
```

Figure 3.3: Pseudo-code of dynamic programming algorithm for finding prefix grams.

3.2.3 Cache-Efficient Filtering of Candidate Mappings

A straightforward implementation of the q-gram-based filter and verification procedure can lead to poor performance due to cache misses. Since the reference sequence is often much larger (e.g., 3GB for the human genome) than the CPU cache (e.g., 12MB L3 cache for an Intel Xeon X5670), each verification of a candidate position likely causes a cache miss. Since most candidate positions typically are false positives, paying a cache miss for fetching an

irrelevant substring of the reference sequence is very wasteful.

In this section, we present a cache- and compute-efficient filter for removing false-positive candidate mappings without accessing the reference sequence. The main idea is to augment the inverted lists with additional filtering data, such that it is in the CPU cache during the traversal of an inverted list.

Mapping q-gram neighborhoods to bitvectors: We attach to each inverted-list element an encoding (discussed next) of its corresponding neighboring characters in the reference sequence using 1 bit per character. The top of Figure 3.4 illustrates this procedure on an exemplary 5-gram at position 112 in a reference sequence. We use 16 bits to encode the 8 characters to the left and right of the 5-gram `ACCCT` using a mapping $A, T \Rightarrow 0$, and $C, G \Rightarrow 1$ to create the bitvector. Both the position 112 and its bitvector $b(112)$ are inserted into ACCCT's inverted list. Since we only access `ACCCT`'s inverted list if `ACCCT` is also contained in the read we are processing, it is unnecessary to include `ACCCT` itself in the bitvector. The size of the bitvectors is a tunable parameter, and we use 16 bits for this example.

It might seem that using a single bit per character reduces the filtering capability by 50% because we map strings of a 4-letter alphabet ($\mathtt{A}, \mathtt{C}, \mathtt{T}, \mathtt{G}$) to a 2-letter alphabet ($\mathtt{0}, \mathtt{1}$). But, it is well known that not every character substitution is equally likely on real data [31]. For example, Table 3.1 shows the frequency of character substitutions we gathered in a simple experiment, as follows. For each of the 35bp reads we computed the optimal gram prefix and traversed the corresponding inverted lists to obtain candidate mapping positions. Next, we recorded the frequency of character substitutions during the verification of these candidate positions. Since "$\mathtt{A} \rightarrow \mathtt{G}$" and "$\mathtt{T} \rightarrow \mathtt{C}$" are the most frequent substitutions, our results suggest the following encoding: $\mathtt{A}, \mathtt{T} \Rightarrow 0$ and $\mathtt{C}, \mathtt{G} \Rightarrow 1$, such that the characters of the most frequent substitutions get different bit values.

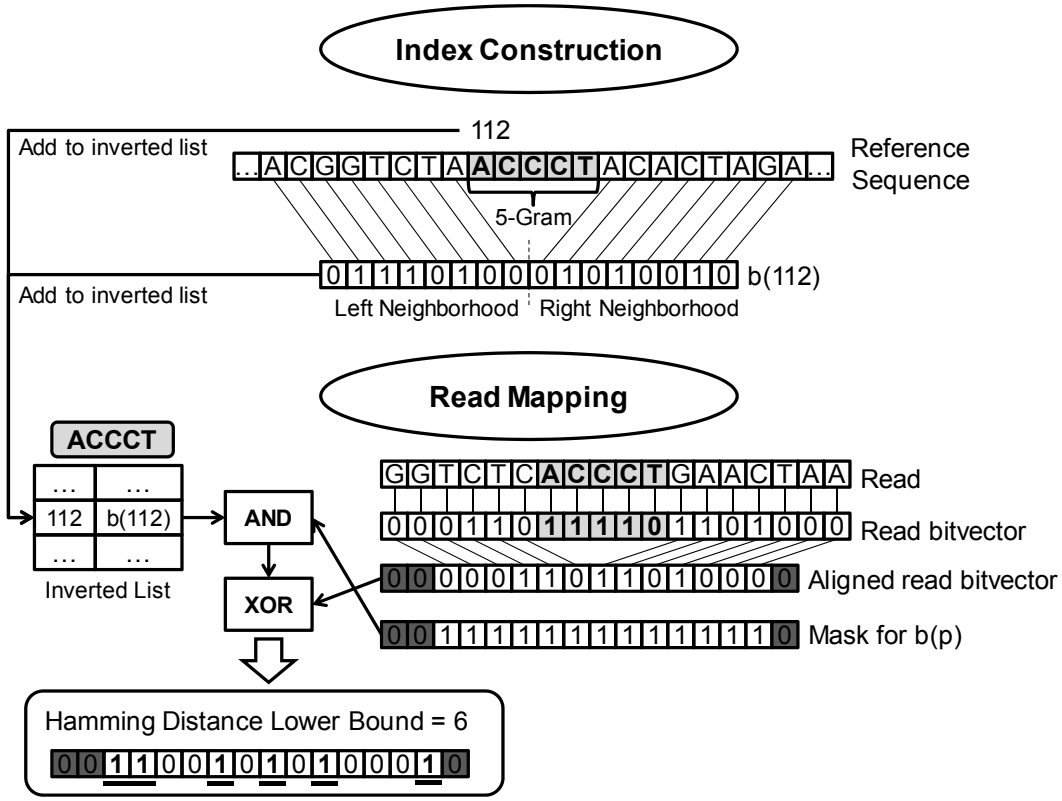


Figure 3.4: Constructing an index with bitvectors (top) and filtering candidate answers (bottom). We use a mapping $A, T \Rightarrow 0$, and $C, G \Rightarrow 1$ to create the bitvectors. The dark gray boxes on the bottom indicate invalid bits we must ignore, based on `ACCCT`'s position in the read `GGTCTCACCCTGAACTAA`. The light gray boxes highlight the matching q-gram `ACCCT`.

Read character	HG18 character	# of substitutions
A	T	687,276,051
A	G	**1,382,950,075**
A	C	559,937,841
T	G	395,839,922
T	C	**1,232,657,183**
G	C	393,616,199

Table 3.1: Frequency of character substitutions using 2 million 35bp reads on HG18. The results suggest a mapping: `A`, `T` $\Rightarrow 0$ and `C`, `G` $\Rightarrow 1$.

Apart from representing more characters with a fixed number of bits, our bitvector encoding also allows CPU-efficient filtering of candidate positions, as follows.

Candidate filtering using bitvectors: Let us revisit the example in Figure 3.2. After computing an optimal gram prefix we need to traverse their corresponding inverted lists to find candidate mappings. Suppose we begin with the list of gram `ACCCT`, since it is the shortest list of those in the optimal prefix.

The bottom of Figure 3.4 illustrates how we use the bitvectors for filtering. Before scanning `ACCCT`'s inverted list, we map the read to a bitvector. Next, we "shift away" the bits of the matching q-gram `ACCCT` in the read's bitvector to align the positions of its bits with those in the reference-sequence. The relevant pseudo-code is shown in Figure 3.5.

Input:
  Read bitvector b_s.
  Bitvector width in bits w.
  Gram length q.
  Position of matching q-gram p.
Output:
  A read's aligned bitvector.
Method: AlignBitvector
1. $M_L = (p-1)$ "1" bits followed by $(|b_s| - p + 1)$ "0" bits;
2. $M_R = (p-1)$ "0" bits followed by $(|b_s| - p + 1)$ "1" bits;
3. $L = (b_s \text{ AND } M_L) >> (w/2 - (p-1))$;
4. $R = (b_s \text{ AND } M_R) << (p + q - 1) - w/2))$;
5. $b = L + R$;
6. **RETURN** First w bits of b;

Figure 3.5: Procedure for aligning a read's bitvector. We align the read's bitvector to the bits in the bitvectors of the reference sequence, based on the position of a matching q-gram in the read.

Recall that we omitted the q-gram itself when generating bitvectors for the reference sequence. This shifting produces invalid bits at both ends of the read bitvector shown as dark boxes. These invalid bits represent portions of the bitvectors from the reference sequence that we cannot use for pruning candidates. For example, since there are only 6 characters to the left of `ACCCT` in the read, we should ignore 2 of the 8 bits representing the left

neighborhood of ACCCT's occurrences in the reference sequence. We generate a bitmask to remove those invalid bits from each bitvector in ACCCT's inverted list. The pseudo-code for generating such a bitmask is shown in Figure 3.6.

Input:
    Read length $|s|$.
    Bitvector width in bits w.
    Gram length q.
    Position of matching q-gram p.
Output:
    Bitmask to remove invalid bits.
Method: GetBitMask
1. $M = w$ "1" bits;
2. $L = w/2 - (p - 1)$;
3. $R = max(0, (p + q - 1) - w/2 - (|s| - w))$;
4. Set first L bits of M to "0".
5. Set last R bits of M to "0".
6. **RETURN** M;

Figure 3.6: Procedure for computing a bitmask for reference-sequence bitvectors. The resulting bitmask removes invalid bits from bitvectors of the reference sequence, based on the position of a matching q-gram in a read.

Now that we have aligned the read's bitvector with the bitvectors in the inverted list, and generated a bitmask to remove invalid bits from those bitvectors, we start scanning ACCCT's inverted list. For each candidate position, we use the read's bitvector and the candidate's bitvector to compute a lower bound on the Hamming distance between their corresponding original sequences, as follows. First, we do a "bitwise-AND" operation between the bitmask and the candidate's bitvector. Then, we do a "bitwise-XOR" operation between the resulting bitvector and the read's bitvector to produce a final bitvector. In the final bitvector, a bit is set to 1 if and only if the original character at the corresponding position in the read is different from the corresponding character in the reference sequence. We prune a candidate if the number of 1-bits in the final bitvector exceeds our Hamming-distance threshold. We determine the number of 1-bits in the final bitvector with a single CPU instruction, `popcount`, supported by most modern CPUs.

In summary, our new bitvector-filtering technique eliminates candidate mappings without accessing the reference sequence, thus avoiding expensive cache misses. In addition, our filter only requires a handful of CPU instructions per inverted-list element, namely bitwise-AND, bitwise-XOR, `popcount`, and a final comparison with the Hamming-distance threshold.

Read mapping procedure: Figure 3.7 shows the complete procedure for finding all mappings of a read s within a Hamming distance d. The procedure employs our new optimizations, namely the optimal prefix grams, and a bitvector-enhanced inverted index. First, we compute the read's bitvector and its optimal set of prefix grams sorted by inverted-list length (lines 1-3). Next, we initialize an empty candidate set and process the inverted lists corresponding to the optimal prefix grams one by one. For each prefix gram, we construct the read's aligned bitvector and the bitmask to be applied to each inverted-list element's bitvector. Each iteration (loop in line 5) produces a new set of candidates by traversing the inverted list corresponding to the gram at position $P[i]$, and sort-merging it against the previous set of candidates to eliminate duplicates. While traversing an inverted list (loop in line 10), we prune candidate positions using our bitvector method. In lines 12-13 we apply the bitvector filter to a new position that is not contained in our set of candidates. On the other hand, we also apply the filter to a position that is already in the set of candidates (lines 21-21) because we may still be able to prune it even though that position had passed a bitvector filter at some point. The reason why a subsequent filtering attempt could be successful is that the aligned bitvector and the bitmask (lines 6-7) depend on the position of the corresponding q-gram in the read. A different q-gram position (in the read) may result in fewer invalid bits that need to be masked away from the inverted-list bitvectors, and ultimately, a tighter lower bound on the Hamming distance. Finally, after processing all relevant inverted lists, we remove false-positive candidate positions by computing the real Hamming distance between the substring starting at that position in the reference sequence and the given read (line 29).

```
Input:
    Gram length q.
    Read s.
    Inverted lists L corresponding to a read s's q-grams G(s).
    Note: Inverted lists contain pairs (p, b) of position and bitvector.
    Hamming-distance threshold d.
    Bitvector width in bits w.
Output:
    A set R of mapping positions within Hamming distance d of read s.
Method: FindMappings
1.  b_s = GetBitvector(s);
2.  P = FindOptimalPrefix(q, L, d);
3.  SortByListLength(P);
4.  C = {}; // candidate mappings
5.  FOR i = 1 TO |P|
6.      b = AlignBitvector(b_s, w, q, P[i]);
7.      m = GetBitMask(|s|, w, q, P[i]);
8.      C_new = {}, k = 1, j = 1;
9.      list = L[P[i]];
10.     WHILE k ≤ |C| OR j ≤ |list| DO
11.         IF k ≤ |C| OR (j ≤ |list| AND C[k] > list[j].p) THEN
12.             IF Popcount(b XOR (list[j].b AND m)) ≤ d THEN
13.                 C_new.add(list[j].p);
14.             END IF
15.             j++;
16.         ELSE IF j ≤ |list| OR (k ≤ |C| AND C[k] < list[j].p) THEN
17.             C_new.add(C[k]);
18.             k++;
19.         ELSE
20.             IF Popcount(b XOR (list[j].b AND m)) ≤ d THEN
21.                 C_new.add(C[k]);
22.             END IF
23.             k++;
24.             j++;
25.         END IF
26.     END WHILE
27.     C = C_new;
28. END FOR
29. R = VerifyCandidates(C);
30. RETURN R;
```

Figure 3.7: Procedure for finding all read mappings within a Hamming-distance threshold.

3.2.4 Supporting Insertions and Deletions

Allowing insertions and deletions (called "indels") is important for mapping longer reads, because both sequencing errors and genetic variations can result in the deletion/insertion of bases and the chance of this happening increases as reads become longer. However, finding mappings with indels is computationally more challenging. Hobbes implements the following two methods for mapping reads with indels.

Hamming distance tends to be sufficient for shorter reads, whereas edit distance becomes important for long reads. Ultimately, the user must decide whether the added benefit of edit distance offsets its computational cost.

Non-heuristic mapping: To find all mappings of a read while allowing indels, we can use the optimal prefix grams as described in the preceding section for generating candidate mapping positions. However, the bitvector filter mentioned above is specific to Hamming distance. A similar filter for indels is possible, and we leave this direction for future work. To verify a candidate position, we conceptually consider those substrings with all possible starting and ending positions (based on the edit-distance threshold), and compute their edit distances to the read. For each candidate position, we report the substring with the lowest edit distance. This approach tends to be very slow, and the following heuristics can significantly improve the mapping performance.

Seed extension approach: Again, we begin with the optimal prefix grams for finding candidate positions. Next, we introduce two heuristics to improve performance. First, we fix the starting position for verification, but shift it to the left once, if the initial position yielded no match. Second, we apply the bitvector filter to the neighborhood of matching grams in the reference sequence, as shown in Figure 3.8.

Since most differences are due to substitutions, our intuition is that if the neighborhood

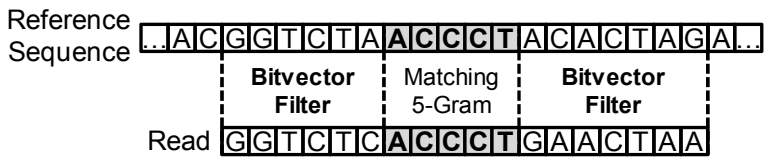


Figure 3.8: Seed extension approach with indels. We prune candidate positions by applying the bitvector filter on the neighborhood of matching grams.

already has a high Hamming distance to the corresponding substring in the read, then the candidate is probably not a match. The filter could remove valid mappings if those apparent substitutions are caused by inconveniently located indels. On the other hand, this effect is mitigated by using multiple grams. It is somewhat unlikely that the neighborhoods of all those grams have a high Hamming distance for true mappings. The bitvector-filter threshold on the neighborhood is a tuning parameter. We found that by setting it to 2/3 of the original edit distance, we capture most mappings while retaining high speed.

Letter count filter: Hobbes uses the letter-count filter described in [62] to quickly detect whether the two sequences can be within a given edit-distance threshold during the verification step. The main idea of the filter is to count the number of occurrences of each character in both sequences, and establish a lower bound on the edit distance between the two sequences using the differences of those character counts, as follows. We divide the frequencies of all base-pairs into two groups. The first group contains the base-pairs that are more common in the read, and the second group contains those that are more common in the candidate. For each group, we create a sum of the frequency-differences of corresponding base-pairs in that group, denoted by Δ_1 and Δ_2, respectively. The grouping ensures that an edit operation can change each Δ by at most 1, establishing $max(\Delta_1, \Delta_2)$ as a lower bound on the edit distance between two sequences. For example, consider sequences $\mathtt{s_1}$ = `AAACCTG` and $\mathtt{s_1}$ = `CCCCTTGG` , $\Delta_1 = (3 - 0) = 3$ (A is more frequent in s_1) and $\Delta_2 = (4 - 2) + (2 - 1) + (2 - 1) = 4$ (C, G, and T are more frequent in s_2). Based on the letter-count filter, a lower bound on the edit distance between s_1 and s_2 is $max(3, 4) = 4$. To prune candidates that survive the

letter-count filter, we also employ standard q-gram counting.

3.2.5 Supporting Paired-End Alignment

Many next-generation sequencing technologies provide the user with *paired-end reads* that contain extra information about the relative position of two reads with respect to each other, as shown in Figure 3.9.

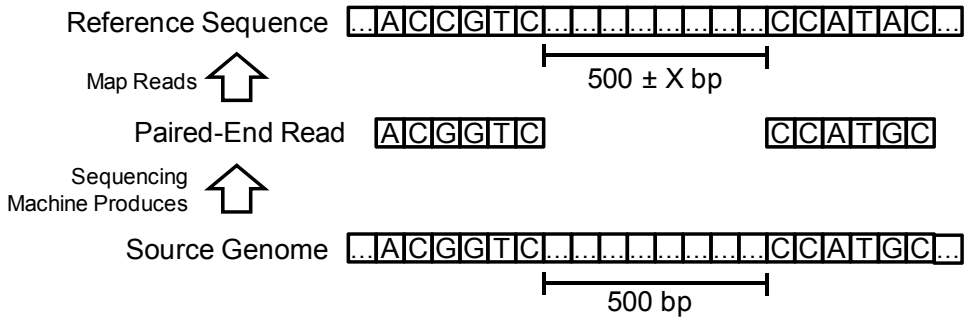


Figure 3.9: An illustration of paired-end reads.

A paired-end read consists of two reads that are know to be a certain distance apart from each other in their originating genome (500bp in the figure) by virtue of the employed sequencing technology. As shown in the figure, when mapping such paired-end reads against a reference sequence we cannot assume that the reads are the same distance apart from each other as they are in their source genome because of genetic variations. Therefore, we must relax this distance constraint when mapping paired-end reads againat a reference sequence. The lower and upper bounds of this distance constraint are often referred to as the *minimum insert size* and *maximum insert size*, respectively. Choosing the distance constraint is generally application dependent, and existing read-mapping packages such as BWA [80] implement estimation procedures to guide users in choosing a reasonable distance constraint (we used BWA for this purpose in our experiments). After mapping paired-end reads, the true distance information provides researchers with additional useful information.

For example, if the paired-end read shown in Figure 3.9 mapped to the reference sequence with a 1000bp distance between the reads of the pair, then it suggests the source genome has a large deletion in that region.

Hobbes supports the alignment of paired-end reads. To align paired-end reads, Hobbes initially considers each read of a pair separately and finds the set of candidate locations for each read. For example, given a read pair (r_1, r_2), Hobbes first finds the candidate location sets C_1 and C_2 corresponding to read r_1 and r_2 respectively. During the next step, for each candidate $c_1 \in C_1$ Hobbes performs verification only if there is a $c_2 \in C_2$ that satisfies the paired-end distance constraint with respect to c_1. Hobbes provides the option of reporting the alignments of each read in a paired-end read separately if no paired alignments are found.

3.2.6 Implementation Details

In this section, we discuss implementation details of Hobbes.

Treatment of `N` characters: Sequencers sometimes produce reads containing `N` characters meaning that the base of that position could not be reliably inferred. We ignore q-grams with `N` characters. As a consequence, our inverted index does not contain those q-grams. When generating q-grams for a read, we may generate fewer than $d + 1$ non-overlapping q-grams, since we ignore q-grams with `N` characters. In our current implementation, we cannot find any mapping for such a read, although we could rely on those other q-grams to find *some* mappings. In all other cases, we treat `N`s as mismatches. Note that we can deal with reads containing `N`s as long as there are enough non-overlapping q-grams without `N`s, and a read can map to substrings in the reference sequence containing `N`s even though the inverted index does not contain q-grams with `N`'s (we may reach such a position via a different, regular q-gram).

Hashing q-grams: We employ a collision-free hash function to map q-grams to integers, as follows. Each character $\{\texttt{A}, \texttt{C}, \texttt{T}, \texttt{G}\}$ is encoded as 2 bits, and the concatenation of all such 2-bits corresponding to a q-gram forms the q-gram's hash code. With this scheme, 32-bit integers can support hashing q-grams up to length $32/2 = 16$, i.e., $q \leq 16$. For longer q-grams we use 64-bit integers.

Hamming-distance verification: Before computing the actual Hamming distance between two sequences using a character-by-character comparison, we do a significantly faster chunk-by-chunk comparison, typically with a chunk size of 4 bytes. If more than d chunks differ, then the two sequences cannot be within Hamming distance d, and we avoid the character-by-character comparison.

Edit-distance verification: After a candidate passes the letter count filter, we compute the real edit distance between two sequences using SeqAn's [36] implementation of Myer's bit-parallel algorithm [92]. The main idea of Myer's bit-parallel algorithm is to compactly represent the columns of the dynamic programming matrix (see Section 2.1) using 2 bits per cell. A key observation is that the value of a particular cell only changes by +1 or -1 (or stays the same) from column to column, therefore, we can represent such differences in cell values sing 2 bits per cell. Myer's algorithm exploits this representation to populate the dynamic programming matrix column by column in batches of rows using bit operations, such that a few bit operations compute a batch of new cells values.

Cache-prefetching during verification: Our bitvector filter can dramatically reduce the number of cache misses by pruning false positives without accessing the reference sequence. However, the number of surviving candidates can still be in the tens of thousands. Once a candidate has passed the bitvector filter, we cannot avoid a cache miss for the distance verification. But we can mitigate the cost by prefetching a future candidate's data into the cache, thus overlapping the verification of the current candidate and the data transfer from memory to cache for the future candidate. We have found that for our CPU architecture

and our set of experiments the best performance is achieved when prefetching the data for candidate number $c+2$ before verifying candidate number c, i.e., we overlap the verification of three candidates with data prefetching.

3.3 Results

3.3.1 Implementation and Setup

We implemented Hobbes in C++, and compiled it with GCC 4.4.3. All experiments were run on a machine with 96GB of RAM, and dual quad-core Intel Xeons X5670 (8 cores total) at 2.93GHz, running a 64-bit Ubuntu OS. Note that Hobbes performs best on CPUs that support the `popcount` instruction. We used GCC's built-in functions for `popcount` and cache prefetching. Hobbes is freely available at http://hobbes.ics.uci.edu, and can output its results in the standard SAM[2] format for analysis with SAMTools.

3.3.2 Other Read Mappers and Data

We compared Hobbes with the following packages. For each package we ran experiments using the same input read files, and if not stated otherwise, we used the SAM output format to ensure their output files containing the mapping results are of similar size. We present the mapping times and number of mappings as reported by each individual package.

Bowtie [71] is a BWT-based short read aligner, and is efficient for finding few mappings per read (1 by default) with a very small memory footprint. Bowtie performs a DFS on the index and stops when the first qualified mapping is found.

[2]http://samtools.sourceforge.net/SAM1.pdf

BWA [80] is also a BWT-based program and supports indels, while Bowtie does not. BWA uses a backtracking search similar to Bowtie's to handle mismatches. By default, BWA adopts an iterative strategy to accelerate its performance, at the price of potentially losing mappings. To report all feasible mappings, we disable the iterative search ("-N" option) in our experiments.

mrsFast [46] and **mrFast-CO** [10] are recent gram-based packages for gapped and ungapped alignment, respectively. They index both the reference genome and the reads. mrsFAST uses an efficient all-to-all list comparison algorithm, while mrFAST-CO follows a seed-and-extend strategy.

RazerS2 [117] builds a gram-based index on the reads, and performs gram counting while scanning over the reference sequence. RazerS2 has been reported to be very accurate in finding all mappings for typical read lengths. We set RazerS2's "max-hit" parameter to 300,000,000 to get all mappings.

Data: We conducted the experiments using reads with 35, 51, 76, and 100 base pairs. The 35bp reads are taken from the YH database [1], the 51bp reads [2] and 76bp reads [3] reads come from the DDBJ DNA Data Bank of Japan (DDBJ) repository with entry DRX000359 and DRX000360 respectively, and the 100bp [4] reads are from specimen HG00096 of the 1000 genome project (http://www.1000genomes.org/data). In all cases, we used the human genome with NCBI HG18 as our reference genome. As we do alignments read by read, the performance of Hobbes is almost linearly proportional to the number of reads. So, in the following, we mainly test the performance of Hobbes and other read mappers using datasets with 500K reads randomly chosen from the above mentioned databases.

3.3.3 Index Construction and Memory Footprint

We used an inverted index of overlapping q-grams on the reference genome. As described earlier, to avoid cache misses, we augmented the inverted lists with bit vectors representing the neighboring characters of the corresponding q-grams. In general, the index size is linearly dependent on the size of the reference sequence and the chosen bit-vector size. By default, Hobbes uses 16-bit vectors, resulting in a total index size of 21GB for HG18. We used bitvector sizes of 16 and 32 bits for our experiments with edit and Hamming distance, respectively. Using a single thread, it took Hobbes 20 minutes to build an index on HG18, whereas Bowtie and BWA needed 114 and 56 minutes, respectively. In addition, Hobbes has a tight-knit multi-threaded framework that parallelizes both indexing and mapping. On multi-core machines, users can build an index as large as HG18 in a few minutes. Because Hobbes does alignment read by read, its memory requirement is independent of the number of input reads.

3.3.4 Results Using Hamming Distance

All mappings: We configured the packages to find all mappings per read. Table 3.2 shows the mapping time, the fraction of reads with at least one mapping, and the total number of mappings for various read lengths and hamming distances. We observe that Hobbes was up to 5 times faster than other packages (even 100 times faster than RazerS2), while producing comparable mappings. For example, on 35bp reads, Hobbes was more than 4 times faster than Bowtie*, which was the fastest among all other listed programs; on 51bp and 76bp reads, Hobbes was about 3 times faster than our closest competitor BWA. Moreover, Hobbes mapped slightly more reads than BWA in that setting. Among the tested programs, mrsFAST and RazerS2 consistenly achieved the best mapping quality. Hobbes delivered a similar quality, while outperforming mrsFAST and RazerS2 in mapping speed by

a factor of up to 10 and 200, respectively. Hobbes achieves such a good performance for the following reasons. First, Hobbes indexes the reference sequence using grams, so finding all mappings per read is simply a matter of traversing a few inverted lists and verifying their candidate locations. Similarly, finding only a few mappings means terminating the inverted-list traversals early. On the other hand, the BWT-based solutions such as Bowtie and BWA are based on a heuristic tree-traversal with backtracking, and obtaining all mappings could require exploring a large portion of the tree. In general, the complexity of such tree-based methods increases with the read length and the number of allowed mismatches, though Bowtie and BWA still achieve a very high speed for many common cases. Compared to the other gram-based approaches, mrsFast and RazerS2, Hobbes performed favorably due to our efficient gram selection and candidate filtering techniques.

Read length (Hamming)	35bp (2 errors)			51bp (3 errors)		
Algorithm	Time (h:m)	Reads mapped (%)	Total mappings (million)	Time (h:m)	Reads mapped (%)	Total mappings (million)
Bowtie*	0:28	76.61	492.6	0:34	91.93	317.1
Bowtie	0:54	76.61	492.6	0:50	91.93	317.1
BWA	0:30	76.61	492.6	0:24	91.61	277.6
mrsFAST	0:43	76.61	492.6	0:59	91.93	317.1
RazerS2	6:38	76.61	488.5	7:58	91.93	316.9
Hobbes	0:06	76.61	492.6	0:08	91.93	317.1
Read length (Hamming)	76bp (3 errors)			76bp (4 errors)		
Bowtie*	0:16	91.44	73.4	NA	NA	NA
Bowtie	0:18	91.44	73.4	NA	NA	NA
BWA	0:10	91.36	71.1	0:18	92.47	115.2
mrsFAST	0:50	91.44	73.4	1:10	92.69	127.2
RazerS2	8:58	91.44	73.4	8:08	92.69	127.2
Hobbes	0:03	91.44	73.4	0:07	92.69	127.2

Table 3.2: Results of mapping 500K single-end reads against HG18. We used Bowtie in its default mode and an optimized mode (Bowtie*) where we set offrate=0 for maximum speed. *Reads mapped*: the fraction of reads with at least one mapping; *Total mappings*: the total number of mapped locations in the reference; *NA:* Bowtie does not support more than 3 mismatches.

A few mappings per read: Some applications may require all mappings per read, and others only a few mappings. Most tools are optimized for only one of those cases. For example, Bowtie focuses on finding a few mappings per read, and is very good at it. Therefore, the comparison in Table 3.2 somewhat disfavors those packages not designed for finding all mappings. To accommodate the few-mappings use case, Hobbes efficiently supports finding any number of mappings. Figure 3.10 shows the mapping times of BWA, Bowtie, and Hobbes for a varying number of requested mappings per read (k). We see that Hobbes performed comparably to Bowtie for very small k, but as k increased Hobbes began to outperform other packages by a growing margin. The reason for this increasing performance gap is the algorithmic difference of the gram-based approaches (hashing and traversing inverted lists) versus the tree-based approaches (heuristic tree-traversal with backtracking).

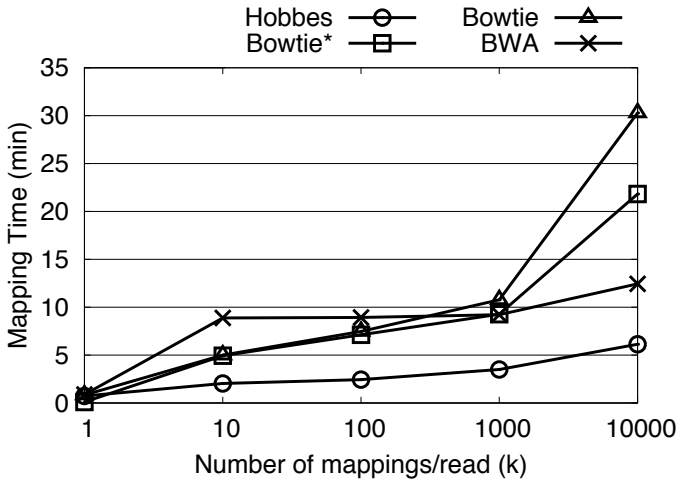


Figure 3.10: Maximun number of mappings k per read *vs.* mapping time. The experiment was run on 51-bp reads with Hamming distance 3. We omitted RazerS2 due to its long mapping time, and mrsFAST because it only supports finding all mappings.

3.3.5 Results Using Edit Distance

Supporting edit distance is significantly more challenging than Hamming distance, and is becoming increasingly important as reads get longer and tend to contain indels. In addition,

many downstream analyses benefit from mapping as many reads as possible to at least one location which may warrant a more expensive mapping procedure allowing indels. Hobbes implements a seed extension approach to align reads within a given edit distance threshold to take advantage of the two optimization strategies we have developed. Although unlike the Hamming distance mapping, the seed extension approach cannot guarantee to find all correct mapping locations, we will show that by using multiple seeds the mapping quality of Hobbes can be achieved to be nearly optimal. Bowtie does not support edit distance, so we compared Hobbes with BWA, mrFast-CO, and RazerS2. We configured the packages to find all mappings per read.

All mappings: Table 3.3 lists our experimental results. We observed that Hobbes was about twice as fast as BWA and mrFast-CO, and over 7 times faster than RazerS2. Notice that the performance gap increased with longer reads and higher edit distances. On 76bp reads, RazerS2 could map slightly more reads than the other packages; however, the mapping took an order of magnitude more time than Hobbes. The speed of mrFast-CO and BWA was similar, and each mapped slightly fewer reads. Hobbes was about twice as fast as mrFast-CO and BWA while mapping more reads. On 100bp reads, RazerS2 had the best quality but a comparably slow mapping speed; BWA became slower than RazerS2 and lost quality at the same time; the quality of Hobbes was very close to that of RazerS2. Compared to mrFast-CO, Hobbes could map more reads and was about 1.5 times as fast. In addition, the current version of mrFast-CO is limited to edit distance 6, which is problematic for mapping even longer reads, while Hobbes does not have such a limitation.

3.3.6 Evaluation on Simulated Data

To more accurately assess the mapping quality of the different programs, we simulated reads from the human genome using the wgsim program [79], and then ran Hobbes to map those

Read length (edit distance)	76bp (4 errors)			100bp (6 errors)		
Algorithm	Time (h:m)	Reads mapped (%)	Total mappings (million)	Time (h:m)	Reads mapped (%)	Total mappings (million)
BWA	02:33	94.06	141.1	22:54	92.16	79.4
mrFast-CO	02:45	94.28	142.3	03:47	92.39	96.3
RazerS2	12:26	94.32	143.6	17:14	92.50	96.4
Hobbes	01:46	94.30	145.8	02:48	92.47	100.7

Table 3.3: Results of mapping 500K single-end reads against HG18.

reads back to the same human genome. By using simulated reads, we know the "correct" mapping positions of those reads based on which we calculate an error rate for the mapping results of different packages. We used the default setting of wgsim, in which the mismatched bases are chosen randomly with a mismatch rate of 2% per base, and 15% of polymorphisms are indels with their sizes drawn from a geometric distribution with mean 1.43.

Since Hobbes is only guaranteed to be exact for Hamming distance, we use the simulated data to examine the mapping quality of Hobbes using edit distance. We use two metrics to measure the accuracy of mapping: one is the fraction of reads with at least one mapping and the other is the mapping error rate. We say a read is aligned correctly if the true location of the read starts at the same location as one of its mappings. The mapping error rate is defined to be the fraction of mapped reads that are aligned incorrectly.

Table 3.4 shows the performance of Hobbes as compared to other programs in the case of edit distance. In terms of speed, Hobbes was the clear winner - about 3.5 times faster than BWA and 6 times faster than RaserS2 for 100bp reads. In terms of the fraction of reads mapped, Hobbes' result was slightly worse than that of the best program, RaserS2, but the margin was small with a difference of only 0.02% for both 76bp and 100bp reads. Hobbes achieved the best mapping error rate of 0.22%. These results demonstrate that although Hobbes sacrifices some accuracy for speed, its mapping quality was comparatively better

than the other packages tested.

Read length (edit distance)	76bp (4 errors)			100bp (6 errors)		
Algorithm	Time (h:m)	Reads mapped (%)	Error rate (%)	Time (h:m)	Reads mapped (%)	Error rate (%)
BWA	01:22	96.05	2.17	07:55	97.09	1.78
mrFast-CO	02:15	97.84	3.43	03:22	99.43	3.63
RazerS2	10:08	97.90	0.98	12:59	99.50	1.15
Hobbes	01:08	97.88	0.22	02:20	99.48	0.22

Table 3.4: Results of mapping 500K simulated reads.

3.3.7 Paired-End Alignment

We compared Hobbes with other state-of-the-art packages using paired-end reads.

Hamming distance: For Hamming distance, Hobbes is guaranteed to find all correct mappings. Table 3.5 summarizes the results of Hobbes and several programs for mapping reads of various lengths and Hamming distance thresholds to the human reference genome. We focus on the speed of mapping since the quality of mapping is similar among different programs. Hobbes was close to Bowtie in the 35bp case, but substantially outperformed Bowtie (11 times faster) when the read length increased to 76bp and the Hamming distance increased to 3 due to the inherent costs of tree-based methods for longer reads and more errors. Moreover, for the 100bp case with 4 errors, Bowtie was unable to provide answers because of too many backtracking steps required in the BWT-based algorithm. Hobbes was 24 times faster than the second-best program, mrsFAST in this case. These results suggest that Hobbes outperformed other programs in the Hamming distance case, especially for long reads while allowing many errors. We attribute Hobbes' advantage over mrsFAST to our gram selection and bitvector-filtering techniques which are especially effective for Hamming distance.

Edit distance: Our performance results are summarized in Table 3.6. The fraction of read pairs that can be aligned to the reference sequence is used as a surrogate of mapping quality. In terms of the fraction of mapped pairs, Hobbes was similar to RaserS2, both of which were significantly better than other programs. In terms of mapping speed, Hobbes was clearly the fastest in all three cases with big margins. It was 22 times faster than BWA, 3 times faster than mrFAST, and 15 times faster than RaserS2 in the 100bp case.

Read length (Hamming)	35bp (2 errors)		76bp (3 errors)		100bp (4 errors)	
Algorithm	Time (h:m)	Mapped pairs(%)	Time (h:m)	Mapped pairs(%)	Time (h:m)	Mapped pairs(%)
Bowtie	0:02	84.58	0:18	80.06	NA	NA
Bowtie*	0:20	84.68	0:25	80.06	NA	NA
mrsFAST	0:42	84.66	0:42	80.06	0:52	83.40
Hobbes	0:04	84.68	0:02	80.06	0:02	83.44

Table 3.5: Results of mapping 250K paired-end reads against HG18. Mapped pairs: the fraction of read pairs mapped with at least one location satisfying the distance and orientation constraint. Bowtie*: running with -y option. Some entries contain `NA` because Bowtie does not support more than 3 mismatches.

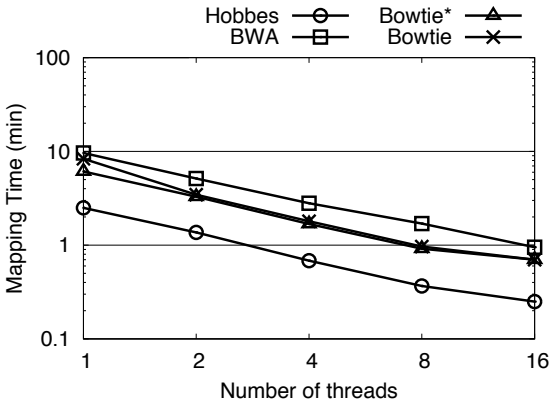
Read length (edit distance)	35bp (2 errors)		76bp (4 errors)		100bp (6 errors)	
Algorithm	Time (h:m)	Mapped pairs(%)	Time (h:m)	Mapped pairs(%)	Time (h:m)	Mapped pairs(%)
BWA*	1:02	84.93	2:15	84.45	22:48	88.14
mrFast-CO	2:32	78.02	2:32	81.61	03:36	85.96
RazerS2	3:57	84.53	10:52	84.49	15:02	88.18
Hobbes	0:26	85.35	0:21	84.60	0:50	88.37

Table 3.6: Results of mapping 250K paired-end reads against HG18. * The raw number of mapped reads output by BWA are higher with 90.76%, 92.60% and 92.66% for 35, 76 and 100 reads, respectively. We removed those mappings that violated the edit distance constraints.

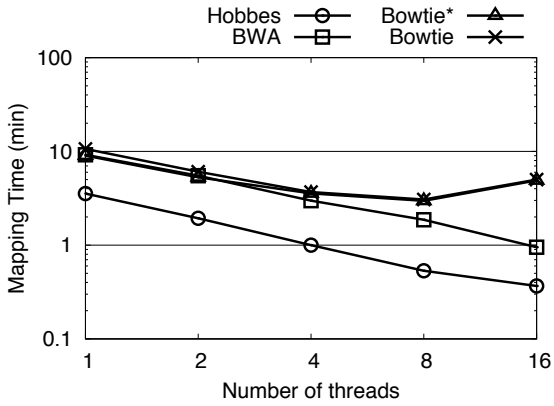
3.3.8 Multi-Threaded Mapping Performance

Due to the current processor trends, it is becoming increasingly important to be able to leverage multiple processor cores effectively. We compared the performance of Hobbes, BWA and Bowtie when using a varying number of threads to map 51bp reads against HG18. We had Hobbes and BWA write their results in the SAM format, and Bowtie in its native format. Both Bowtie and Hobbes are easily parallelizable by distributing the input reads among multiple threads, and then having each thread map its reads in parallel (reads are independent and the indexes are read-only). BWA allows a similar strategy, but currently only supports multi-threading in one of the steps of its mapping procedure. Figure 3.11 shows the mapping times as we increased the number of threads. Notice that the figure does not show mrsFast and RazerS2 because those packages do not support multi-threading (we could not find it in their manual). Hobbes scaled very well when asked to find 100 (a), 1000 (b), or all (c) mappings per read. BWA's relative speed-up became worse as the number of requested mappings per read increased. One explanation could be that BWA only supports multiple threads in one step of their mapping procedure, and as the number of requested mappings increased the non-parallelizable step became significantly more expensive. Notice that Bowtie's mapping time increased beyond 8 threads. We conjecture that Bowtie's speed in this scenario was limited by the disk bandwidth for writing the output file, because we noticed heavy disk usage during the experiment. This observation is consistent with the fact that Bowtie's native output format requires significantly more space than the SAM output format used for BWA and Hobbes. While we arguably rendered our comparison unfair against Bowtie by choosing its native output format, our results do suggest that users should choose their output format wisely. As the number of cores in modern CPUs increase dramatically, disk usage may become a performance bottleneck even when asking for only a few mappings per read. We have shown that Hobbes utilizes multiple threads well, and still outperforms BWA and Bowtie. We note that BWA and Bowtie were not designed for

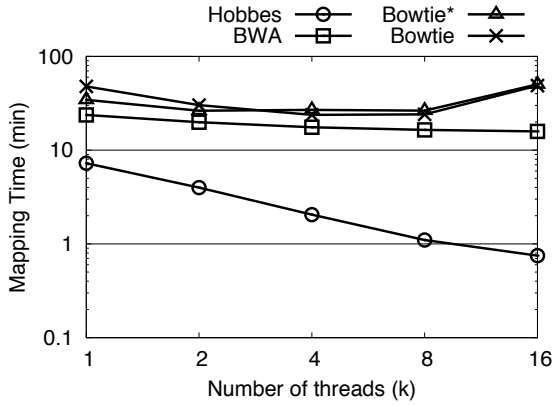
finding all mappings per read, and therefore, the dramatic performance difference between them and Hobbes is not surprising.



(a) Output at most 100 mappings per read.



(b) Output at most mappings 1000 per read.



(c) Output all mappings per read.

Figure 3.11: Multi-threaded mapping times of Bowtie, BWA and Hobbes. We mapped 51bp reads against HG18 allowing up to 3 errors (Hamming distance). mrsFast, and RazerS2 do not support multi-threading (we could not find it in their manual).

3.3.9 Application in RNA-seq abundance analysis

To show the importance of finding all mappings as opposed to only a few, we performed a transcript abundance analysis. In RNA-seq applications it is important to find all mapping positions of a read for quantifying the expression level of a particular gene isoform, due to the occurrence of homologous genes and multiple RNA isoforms originating from the same

gene. We used the UCSC genes on the mm9 mouse (NCBI build 37) assembly as target transcripts; both 76bp paired-end RNA-seq reads were downloaded from Gene Expression Omnibus (Series GSE20846). We compared Bowtie and Hobbes for finding all mappings. The results in Table 3.7 show that Hobbes was 17-20x faster than Bowtie, and could even map slightly more reads.

Reads	SRR047951 (21.8 million)			SRR047953 (20 million)		
Algorithm	Time (h:m)	Reads mapped (%)	Total mappings (millions)	Time (h:m)	Reads mapped (%)	Total mappings (millions)
Bowtie	09:53	40.28	18.399	09:12	42.97	18.883
Hobbes	00:35	40.29	19.157	00:27	42.99	19.393

Table 3.7: Results of mapping 76bp RNA-seq reads against known mouse transcripts. There were a total 55,419 mouse transcripts. To perform the mapping we used Hamming distance 3 and a minimum and maximum insert size of 76bp and 800bp, respectively.

Next, we piped the result of Hobbes directly to eXpress [5], a streaming tool for quantifying the abundances of a set of target sequences from sampled subsequences. eXpress estimates the relative abundance by computing the fragments per kilobase per million (FPKM) value, which could indicate how reliable the mapping results are for downstream analyses. Intuitively, one may think of the FPKM value as capturing how many reads "hit" the bases of our target transcripts, normalized by the length of the transcripts. This "hit rate" is an estimator for the gene expression levels in the sample that our reads originated from, and such gene expression levels are used for determining abnormal cell behaviors which may indicate diseases, etc. We compared the FPKM values when finding at most $k = \{1, 10, 100\}$ mappings, with the FPKM when finding all mappings. To quantify their variation, we computed the ratio of the k-mappings FPKM to the all-mapppings FPKM (or its reciprocal). The results are shown in Table 3.8. We found that among 55,419 target transcripts, the percentage of transcriptions whose ratio was above 1.5 for $k = \{1, 10, 100\}$ was 42.39%, 0.92%, and 0.07% respectively; there was a lot of variability for $k = 1$, meaning the corresponding FPKM value is a poor estimator. The variability reduced as we increased k to 100.

	Transcriptions above ratio (%)	
k vs. All FPKM Ratio	1.5	1.2
k=1 vs. All	42.39	47.12
k=10 vs. All	00.92	01.42
k=100 vs. All	00.07	00.18

Table 3.8: Transcripts with FPKM ratio above 1.5 and 1.2 on 76bp RNA-seq reads. We used Hamming distance 3 to map against 55,419 known mouse transcripts.

The corresponding scatter plots in Figure 3.12 further illustrates this point. We plotted the target transcripts' FPKM values as points, where the x-component of each point is the FPKM value generated in the all-mappings mode and the y-component is the FPKM value generated in the $k = 1, 10, 100$ modes. A "good" value for k should result in a plot very close to a line through the origin, i.e., the x-component is equal to the y-component for all points indicating identical FPKMs for the all-mappings and k-mappings cases. The first figure (upper left) shows that $k = 1$ has significant differences to the all-mappings FPKM values, and this variability reduces rapidly as k increases as shown in the remaining figures. We conclude that Hobbes' strength in finding many mappings per read is indeed a useful feature to ensure meaningful downstream analyses.

3.4 Conclusion

We have presented Hobbes, a local alignment method for mapping short DNA reads to a know reference sequence. Hobbes efficiently supports Hamming and edit distance while finding all mappings or few mappings per read. Our experiments have shown Hobbes to be just as accurate but significantly faster than current read mapping programs.

Hobbes has a large memory footprint (26GB in our experiments), but we believe its speed and mapping quality outweigh that drawback, especially considering today's low memory prices. One of the future directions is to reduce Hobbes memory requirement, possibly via

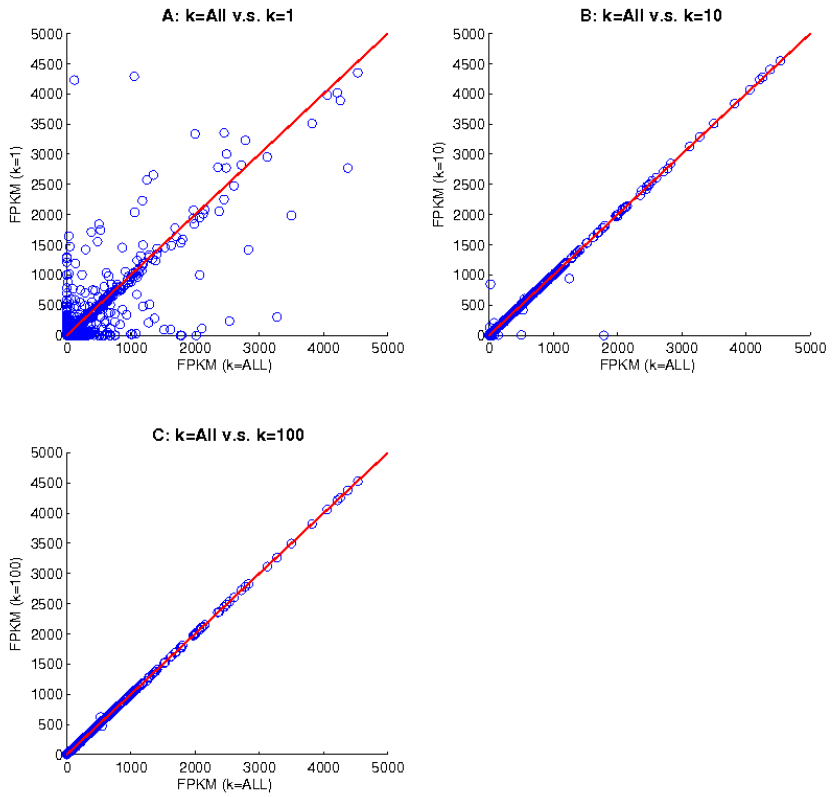


Figure 3.12: Scatter plot of FPKM values when returning at most k mappings. We ran Hobbes and plotted the FPKM values when returning k mappings versus all mappings. In subfigure A, there is a lot of variability when k=1; in subfigure B, the amount of variablity reduces when k=10; in subfigure C, when k=100, the estimated FPKM value become more close to that estimated based on all mapping locations.

compression, or by partitioning our index and performing the read mapping one partition at a time (mrsFast and mrFast-CO follow this approach), as future work.

Given today's trend towards massively multi-core CPUs, we believe that good multi-thread support is of paramount importance. Both Hobbes' index-construction and read-mapping procedures are parallelizable and we have demonstrated that Hobbes scales well with multiple threads. Some packages such as mrsFast, mrFast-CO, and RazerS2 do not support multi-threading at all (we could not find this feature in their manuals). Other packages have certain limitations, e.g., in BWA only one of the two steps during read mapping is parallelizable.

Chapter 4

Compressed Inverted Indexes for Answering Similarity Queries

4.1 Introduction

Gram-based inverted-list indexing structures for global alignment are "notorious" for their large size relative to the size of their original string data. This large index size causes problems for applications. For example, many systems require a very high real-time performance to answer a query. This requirement is especially important for those applications adopting a Web-based service model. Consider online spell checkers used by email services such as Gmail, Hotmail, and Yahoo! Mail, which have millions of online users. They need to process many user queries each second. There is a big difference between a 10ms response time versus a 20ms response time, since the former means a throughput of 100 queries per second (QPS), while the latter means 50 QPS. Such a high-performance requirement can be met only if the index is in memory. In another scenario, consider the case where these algorithms are implemented inside a database system, which can only allocate a limited amount

of memory for the inverted-list index, since there can be many other tasks in the database system that also need memory. In both scenarios, it is very critical to reduce the index size as much as we can to meet a given space constraint.

Contributions: In this chapter we study how to reduce the size of such index structures for the global alignment version of string-similarity search, while still maintaining a high query performance. In Section 4.3 we study how to adopt existing inverted-list compression techniques [126] to our setting. That is, we partition an inverted list into fixed-size segments and compress each segment with a word-aligned integer coding scheme. To support fast random access to the compressed lists, we can use synchronization points [94] at each segment, and cache uncompressed segments to improve query performance. Most of these compression techniques were proposed in the context of information retrieval, in which conjunctive keyword queries are prevalent. In order to ensure correctness, lossless-compression techniques are usually required in this setting.

The setting of approximate string search is unique in that a candidate result must occur only a *certain number* of times among all the inverted lists, and not necessarily on all the inverted lists. We exploit this unique property to develop two novel approaches for achieving the goal. The first approach is based on the idea of discarding some of the lists. We study several technical challenges that arise naturally in this approach (Section 4.4). One issue is how to compute a new lower bound on the number of common grams (whose lists are not discarded) shared by two similar strings, the formula of which becomes technically interesting. Another question is how to decide lists to discard by considering their effects on query performance. In developing a cost-based algorithm for selecting lists to discard, we need to solve several interesting problems related to estimating the time spent in different portions of the query-answering procedure. For instance, one of the problems is to estimate the number of candidates that share certain number of common grams with the query. We notice that several related techniques [61, 87, 72] cannot be used directly to solve these

problems (see Section 4.4.1.4). We develop a novel algorithm for efficiently and accurately estimating this number. We also develop several optimization techniques to improve the performance of this algorithm for selecting lists to discard.

The second approach is combining some of the correlated lists (Section 4.5). This approach is based on two observations. First, the string ids on some lists can be correlated. For example, many English words that include the gram "`tio`" also include the gram "`ion`". Therefore, we could combine these two lists to save index space. Each of the two grams shares the union list. Notice that we could even combine this union list with another list if there is a strong correlation between them. Second, recent algorithms such as [76, 47] can efficiently handle long lists to answer approximate string queries. As a consequence, even if we combine some lists into longer lists, such an algorithm can still achieve a high performance. We study several technical problems in this approach, and analyze the effect of combining lists on a query. Also, we exploit a new opportunity to improve the performance of existing list-merging algorithms. Based on our analysis we develop a cost-based algorithm for finding lists to combine.

We have conducted extensive experiments on real datasets for the list-compression techniques mentioned above (Section 5.5). While existing inverted-list compression techniques can achieve compression ratios up to 60%, they considerably increase the average query running time due to the online decompression cost. Our two novel approaches are orthogonal to existing inverted-list-compression techniques, and offer unique optimization opportunities for improving query performance. Note that using our novel approaches we can still compute the *exact* results for an approximate query without missing any true answers. The experimental results show that (1) the novel techniques can outperform existing compression techniques, and (2) the new techniques provide applications the flexibility in deciding the tradeoff between query performance and indexing size. An interesting and surprising finding is that while we can reduce the index size significantly (up to a 60% reduction) with tolerable

performance penalties, for 20-40% reductions we can even improve the query performance compared to the original index. Our techniques work for commonly used functions such as edit distance, Jaccard, and cosine. We mainly focus on edit distance as an example for simplicity.

4.1.1 Related Work

In the field of compression, many generic algorithms [88, 125] have been developed to compress a list of integers using encoding schemes such as LZ [125], Huffman codes [54], Elias Gamma codes [38], and Bloom filters [19]. Due to the importance of inverted indexes, there have been many inverted-list specific compression methods mostly based on delta-encoding [126, 12, 127, 73, 22]. In Section 4.3 we discuss in more detail how to adopt these existing compression techniques to our setting. One observation is that these techniques often need to pay a high cost of increasing query time, due to the online decompression operation, while our two new methods could even reduce the query time. In addition, the new approaches and existing techniques can be integrated to further reduce the index size, as verified by our initial experiments.

A closely related study [53] discusses how to judiciously select q-grams for answering LIKE queries. Its goal is to maximize the utility of a q-gram inverted index to optimize a given LIKE-query workload on a given dataset, subject to a space constraint on the index. While that study has similarity to our work, there are also key differences. First, our T-occurrence problem (Section 2.4) complicates the matter of determining the utility of choosing a q-gram. For an individual LIKE-query, a q-gram can prune all string ids that are not on the q-gram's inverted list. However, in our T-occurrence problem a string id can only be pruned if it is absent from a certain number of inverted lists, depending on the query, the similarity function and threshold. This property renders it difficult to adapt the graph theoretic formulation

from [53] to our setting, because it relies on the notion that a single gram can prune results from a query (a gram "enables" paths from a query to candidate result in the graph, but in our case we need at least a few grams to enable *any* of those paths). In our work, we also consider fusing inverted lists to save space at the expense of additional false positives. This possibility is not explored in [53] but is equally applicable to LIKE queries. Further, we also make independent contributions specific to our setting, such as the development of a tigher T-occurrence bound for edit-distance queries on our compressed index.

Recently a new technique called VGRAM [77, 123] was proposed to use variable-length grams to improve approximate-string query performance and reduce index size. This technique, as it is, can only support edit distance, while the techniques presented in this chapter support a variety of similarity functions. Our techniques can also provide the user the flexibility to choose the tradeoff between index size and query performance, which is not provided by VGRAM. Our experiments show that our new techniques outperform VGRAM, and potentially they can be integrated with VGRAM to further reduce the index size (Sections 4.6.4 and 4.7).

4.2 Preliminaries

Several existing algorithms [76, 104] have been proposed for answering approximate string queries efficiently. They first solve the T-occurrence problem (Section 2.4.1) to get a set of string candidates, and then check their real distance to the query string to remove false positives. Note that if the threshold $T \leq 0$, then the entire data collection needs to be scanned to compute the results. We call it a *panic case*. One way to reduce this scan time is to apply filtering techniques. For example, length filtering and prefix filtering [44, 76] can separate the collection into different groups. Given a query, we only need to scan some of groups to find the results. To summarize, we must consider the following two cases when estimating the time of answering a query:

- If the lower bound T (also called "merging threshold") is positive, then the time includes the time to traverse the lists of the grams in the query to find candidates (called "merging time") and the time to remove the false positives from the candidates (called "post-processing time").
- If the lower bound T is zero or negative, we need to spend the time (called "scan time") to scan the entire data set, possibly using filtering techniques.

In the following sections we adopt existing techniques and develop new techniques to reduce this index size. For simplicity, we first focus on the edit distance function. In Section 4.7 we will see that the results can be naturally extended to other commonly used functions.

4.3 Adopting Existing Compression Techniques

There are many techniques [126] focusing on inverted-list compression in the literature, which mainly study the problem of representing integers on inverted lists efficiently to save storage space and disk I/O costs. In this section we will briefly review these techniques, adopt them to our problem setting, and then discuss the limitations of these techniques.

4.3.1 Encoding and Decoding Lists of Integers

Most of the list-compression techniques exploit the fact that ids on an inverted list are monotonically increasing integers. For example, suppose we have $l = (id_1, id_2, \ldots, id_n)$, $id_i < id_{i+1}$ for $1 \leq i < n$. If we take the differences of adjacent integers to construct a new list $l' = (id_1, id_2 - id_1, id_3 - id_2, \ldots, id_n - id_{n-1})$, the integers on this new list tend to be smaller than the original ids. This l' list is called the gapped representation of the inverted list l, since it stores the "gap" between each pair of adjacent integers. Using this gapped form, many integer-compression techniques such as gamma codes, delta codes, and Golomb

codes [126] can efficiently encode inverted lists by using shorter representations for small integers. We choose one of the recent techniques called Carryover-12 [12] to adopt in our setting.

There exist two different catagories of compression techniques regarding how data is aligned: bitwise codes and bytewise codes (or word-aligned codes). Bitwise codes pack compressed integer bits tightly into a bit stream while bytewise codes and word-aligned codes align compressed integer bits to the boundary of bytes, or 4-byte integer words. The latter may not have a compression ratio as high as bitwise codes, but decoding them is faster, since aligned integers are much easier to process on CPUs. In our setting, performance is very important and therefore we choose one of the recent word-aligned encoding techniques called Carryover-12 [12], which was shown to be faster than many existing list-compression techniques.

4.3.2 Segmenting, Compressing, and Indexing Inverted Lists

An issue arises when using the encoded, gapped representation of a list. Since decoding is usually achieved in a sequential way, a sequential scan on the list might not be affected too much. However, random accesses could become expensive. Even if the compression technique allows us to decode the desired integer directly, the gapped representation still requires restoring of all preceding integers. Many efficient list-merging algorithms such as DivideSkip [76] rely heavily on binary search on the inverted lists. This problem can be solved by segmenting the list and introducing *synchronization points* [94]. Each segment is associated with a synchronization point and decoding can start from any synchronization point, so that only one segment needs to be decompressed in order to read a specific integer. One way to adopt this technique is to have each segment contain the same number of integers. Since different encoded segments could have different sizes, it is necessary to index the starting offset of each encoded segment, so that they can be quickly located and decompressed.

Figure 4.1 illustrates the idea of segmenting inverted lists and indexing compressed segments.

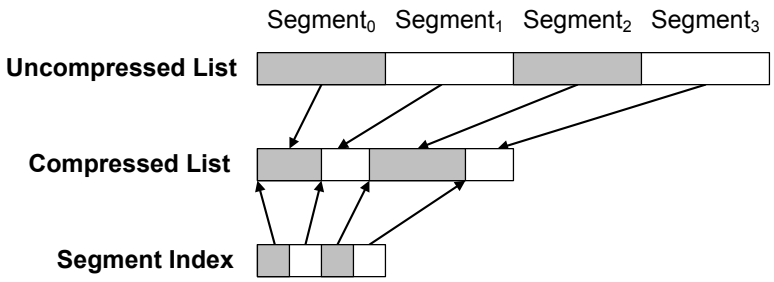


Figure 4.1: Inverted-list compression with segmenting and indexing.

A simple way to access elements is by decoding the corresponding segment for each integer access. If multiple integers within the same segment are requested, the segment will be decompressed multiple times. The repeated efforts can be alleviated using caching. Once a segment is decoded, it will remain in the cache for a while. All integer accesses to that segment will be answered using the cache without decoding the segment again, until the segment is replaced from the cache. We allocate a global cache pool for all inverted lists. More details are given in the description of our experiments (Section 5.5).

4.3.3 Limitations of Existing Techniques

Most of these existing techniques were initially designed for compressing disk-based inverted indexes. Using a compressed representation, we can not only save disk space, but also decrease the number of disk I/Os. Even with the decompression overhead, these techinques can still improve query performance since disk I/Os are usually the major cost. When the inverted lists are in memory, these techinques require additional decompression operations, compared to non-compressed indexes. Thus, the query performance can only decrease. Another limitation is that these approaches have limited flexibility in trading query performance with space savings. Next we propose two novel methods to reduce the size of inverted lists while retaining good query performance.

4.4 Compression Method 1: Discarding Inverted Lists

In this section we study how to reduce the size of an inverted-list index by discarding some of the lists. That is, for all the grams from the strings in S, we only keep inverted lists for *some* of the grams, while we do not store those of the other grams. A gram whose inverted list has been discarded is called a *hole gram*, and the corresponding discarded list is called its *hole list*. Notice that a hole gram is different from a gram that has an empty inverted list. The former means the ids of the strings with this gram are not stored in the index, while the latter means no string in the data set has this gram.

We study the effect of hole grams on query answering. In Section 4.4.1 we analyze how they affect the merging threshold, the list merging and post-processing, and discuss how the new running time of a single query can be estimated. For a single query, we present a dynamic programming algorithm for computing a tight lower bound on the number of nonhole grams shared by two similar strings assuming the edit distance function. Based on our analysis, we propose an algorithm to wisely choose grams to discard in the presence of space constraints while retaining efficient processing. We develop various optimization techniques to improve the performance of this algorithm (Section 4.4.2).

For a particular application it may not be obvious how to choose a reasonable space constraint. Users could decide the space constraint based on the available main memory, or based on an acceptable tradeoff between memory consumption and query performance. Our experimental results in Section 5.5 provide some guidance to deciding this tradeoff. While, in general, picking a space constraint may be difficult, our solutions could still be preferable to alternative compression schemes which provide no guarantee on the resulting index size. In the following, we assume a space constraint as an input to our solutions.

4.4.1 Effects of Hole Grams on a Query

For an approximate query with a string s if $G(s)$ does not have hole grams, then the query is unaffected. If it does have hole grams, then the query may be affected in the following ways:

- The lower bound on the number of nonhole grams shared by a similar string in the collection could become smaller.
- As a consequence, the merging time and post-processing time could both increase.
- However, due to having a lower number of elements among the lists to process, the merging time could also decrease. Also, in some cases the modified threshold could decrease the post-processing time.
- If the new lower bound becomes zero or negative, then the query can only be answered by a scan.

4.4.1.1 Effects on Merging Threshold

Consider a string r in the collection S such that $ed(r, s) \leq k$. For the case without hole grams, r needs to share at least $T = (|s|+q-1)-k\times q$ common grams in $G(s)$ (Equation 2.4). To find such an r, in the corresponding T-occurrence problem, we need to find string ids that appear on at least T lists of the grams in $G(s)$.

If $G(s)$ does have hole grams, the id of string r could have appeared on some of the hole lists. We do not know on how many hole lists this string r could appear, since these lists have been discarded; we can only rely on the lists of those nonhole grams to find candidates. Thus, the problem becomes deciding a lower bound on the number of occurrences of string r on the *nonhole* gram lists.

One simple way to compute a new lower bound is the following. Let H be the number of

hole grams in $G(s)$, where $|G(s)| = |s| + q - 1$. Thus, the number of nonhole grams for s is $|G(s)| - H$. In the worst case, every edit operation can destroy at most q nonhole grams, and k edit operations could destroy at most $k \times q$ nonhole grams of s. Therefore, r should share at least the following number of nonhole grams with s:

$$T' = |G(s)| - H - k \times q. \tag{4.1}$$

We can use this new lower bound T' in the T-occurrence problem to find all strings that appear at least T' times on the nonhole gram lists as candidates.

The following example shows that this simple way to compute a new lower bound is pessimistic, and the real lower bound could be tighter. Consider a query string $s =$ `irvine` with an edit-distance threshold $k = 2$. Suppose $q = 3$. Thus the total number of grams in $G(s)$ is 8. There are two hole grams `irv` and `ine` as shown in Figure 4.2. Using the formula above, an answer string should share at least 0 nonhole gram with string s, meaning the query can only be answered by a scan. This formula assumes that a single edit operation could destroy at most 3 grams, and two edit operations destroy at most 6 grams. However, a closer look at the positions of the hole grams tells us that a single edit operation can destroy at most 2 nonhole grams, and two edit operations can destroy at most 4 nonhole grams. Figure 4.2 shows two deletion operations that can destroy the largest number of nonhole grams, namely 4. Thus, a tighter lower bound is 2 and we can avoid the panic case. This example shows that we can exploit the positions of hole grams in the query string to compute a tighter threshold in the T-occurrence problem. Next, we develop a dynamic programming algorithm to compute a tight lower bound on the number of common nonhole grams in $G(s)$ an answer string needs to share with the query string s with an edit-distance threshold k. A similar idea is also adopted in an algorithm in [123] in the context of the VGRAM technique [77].

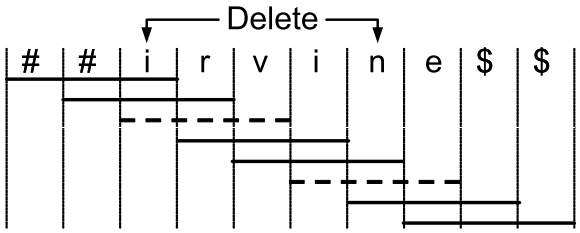


Figure 4.2: A query string `irvine` with two hole grams. A solid horizontal line denotes a nonhole gram, and a dashed line denotes a hole gram.

Subproblem: Let $0 \leq i \leq |s|$ and $1 \leq j \leq k$ be two integers. Let $P(i, j)$ be an upper bound on the number of grams that can be destroyed by j edit operations that are at positions *no greater than* i. The overall problem we wish to solve becomes $P(|s|, k)$.

Initialization: Let $D_i^{d/s}$ denote the maximum possible number of grams destroyed by a deletion or substitution operation at position i, and let D_i^{ins} denote the maximum possible number of grams destroyed by an insertion operation after position i. For each $0 \leq i \leq |s|$ we set $P(i, 0) = max(D_i^{d/s}, D_i^{ins})$.

Recurrence function: Consider the subproblem of computing a value for entry $P(i, j)$. Let g_i denote the gram starting from position i, which may either be a hole or a nonhole gram. If it is a hole then we can set $P(i, j) = P(i - 1, j)$ because an edit operation at this position cannot be the most destructive one. Recall that we have already discarded the list belonging to the gram at i. If the gram starting from i is not a hole then we need to distinguish three cases:

- We have a deletion/substitution operation at position i. This will destroy grams up to position $i - q + 1$ and consume one edit operation. The number of grams destroyed is $D_i^{d/s}$. Therefore, we can set $P(i, j) = P(i - q, j - 1) + D_i^{d/s}$.
- We have an insertion operation after position i. This will destroy grams up to position $i - q + 2$ and consume one edit operation. The number of grams destroyed is D_i^{ins}. Therefore, we can set $P(i, j) = P(i - q + 1, j - 1) + D_i^{ins}$.

- There is no operation at i. We can set $P(i,j) = P(i-1,j)$.

The following is a summary of the recurrence function:

$$P(i,j) = max \begin{cases} g_i \text{ is a hole} : P(i-1,j) \\ g_i \text{ is a gram} : \begin{cases} \text{delete/substitute } i : P(i-q,j-1) + D_i^{d/s} \\ \text{insertion after } i : P(i-q+1,j-1) + D_i^{ins} \\ \text{noop } i : P(i-1,j) \end{cases} \end{cases} \tag{4.2}$$

After we have computed the maximun number of grams destroyed by k edit operations (denoted by D_{max}), we can get the new bound using the same idea as in Equation 2.4, except that the maximum number of grams destroyed is not $k \times q$ but D_{max}.

4.4.1.2 Effects on List-Merging

The running time of some of the merging algorithms (e.g., HeapMerge and ScanCount [76]) is insensitive to the merging threshold T and mainly depends on the total number of elements in all inverted lists. Therefore, the running time for these algorithms can only decrease by discarding some of the lists. Other merging algorithms (e.g., MergeOpt and DivideSkip [76]) separate the inverted lists into two groups, which are processed separately: long lists and short lists. These algorithms perform a heap-based merging on the short lists to get candidate string ids that might occurr T times on all lists. In order to get the final solution to the T-occurrence problem, the algorithms verify those candidates against the long lists using binary search. The performance of these algorithms depends on which lists are put into the group of short list and long lists. For the algorithms mentioned here, the number of long lists is some function of T. Hence, their performance is sensitive to changes in T. In general, by discarding lists their performance may be affected positively or negatively. Which one

is the case depends on the function for separating the lists into groups, and on the choice of lists to discard, i.e. the discarded lists may have belonged to the group of long lists or short lists. Another class of algorithms utilizes T to skip elements on the inverted lists, that cannot be answers to the query. For example, based on T the MergeSkip and DivideSkip [76] algorithms perform a binary search to locate the next element on a list that could potentially be an answer to the query, skipping irrelevant elements. For these algorithms a higher T supports the skipping of elements. Therefore, decreasing T by discarding some lists can negatively affect their performance. However, by discarding some lists conceptually we skip all elements on those lists which may overcompensate for the decreased T, leading to a performance improvement. Note that the DivideSkip algorithm is sensitive to T in two ways: (1) because it uses the concept of short lists and long lists, and (2) because it uses T to skip irrelevant elements.

4.4.1.3 Effects on Post-Processing

Intuitively, for a given query, introducing hole grams may only increase the number of candidates to post-process. We will show that this intuition is correct *if* we use the simple formula for calculating the new threshold T'. Surprisingly, if we use the dynamic programming algorithm to derive a tighter T', then the number of candidates for post-processing can even decrease.

Take the example given in Figure 4.2. Suppose the edit distance threshold $k = 2$. Say that some string id i only appears on the inverted lists of `irv` and `ine`. Since $T = 2$, it is a candidate result. If we choose to discard the grams `irv` and `ine` as shown in Figure 4.2, then each edit operation can destroy at most two non-hole grams and therefore $T' = T = 2$. After discarding the lists, the string i is not a candidate anymore since all the lists containing it have been discarded. This reduces the cost for postprocessing. Note that any string id which appears only on `irv` and `ine` cannot be a result to the query and would have been

removed from the results during post-processing. We will now formalize the above notions. Consider a query with string s, allowing k edit operations, a bag of grams $G(s)$, a bag of hole-grams $H \subseteq G(s)$, and an original merging threshold T.

Lemma 3. *If Equation 4.1 is used to compute the new merging threshold, introducing hole grams cannot decrease the number of candidates for postprocessing.*

Proof. Let us examine a string id that occurred e times on all lists before discarding. If the string was a candidate before, then $e \geq T$. We may have discarded $\delta \leq |H|$ lists on which e occurred. Therefore, $e - \delta \geq T - |H|$, and the string will still be a candidate. If the string was not a candidate before, then $e < T$. The formula $e - \delta < T - |H|$ is not satisfiable for all $\delta \leq |H|$, and therefore the string may become a new candidate. Intuitively, consider the case where we discarded $|H|$ lists on which e did not occur. It may be that $e \geq T - |H|$ although $e < T$. □

Lemma 4. *If the dynamic programming algorithm in Equation 4.2 is used to compute the new merging threshold, introducing hole grams could sometimes decrease the number of candidates for postprocessing. The example given above ilustrates this observation and we shall now generalize the idea.*

Proof. Due to favorable positions of the hole grams it may be that a tighter bound yields $T' \geq T - |H|$ and therefore $T - |H| \leq T' \leq T$. A string id that occurred e times before discarding with $e \geq T$ may cease to be a candidate after introducing hole grams. This is because we may have discarded $\delta \leq |H|$ lists containing e, and if $\delta > e - T$ then $e - \delta < T' \leq T$. □

4.4.1.4 Estimating the Performance of a Query with Hole Grams

Since we are evaluating whether it is a wise choice to discard a specific list l_i, we would like to know, by discarding list l_i, how the performance of a single query Q will be affected using the indexing structure. In the previous sections we have analyzed the effects of introducing hole grams on a query. We now quantify these effects by estimating the running time of a query with hole grams.

Estimating the Merging Time: If the new merging threshold $T' > 0$, we need to solve the T-occurrence problem by accessing the inverted lists of the nonhole grams to find string ids that appear at least T' times. Here, we will consider one of the most efficient merging algorithms, DivideSkip [76], which is sensitive to changes in T. Let the lists $l_1, l_2, \ldots, l_N$ be sorted in a decreasing order of their size. Then the running time for DivideSkip can be estimated as:

$$C_1 * \frac{\sum_{j=L+1}^{N} |l_j|}{T-L} * \sum_{i=1}^{L} log|l_i| + C_2 * \sum_{j=L+1}^{N} |l_j| * log(N-L). \tag{4.3}$$

For $C_1 = 1$ and $C_2 = 1$, the above equation represents the worst cost for the DivideSkip algorithm. The right side of the sum shows the cost of merging the short lists. The left side of the sum presents the cost for verifying the candidates from the short lists against the long lists. The coefficient C_1 represents the fact that the actual number of candidates to verify against the long lists can be smaller than the maximim possible one (given in $\frac{\sum_{j=L+1}^{N} |l_j|}{T-L}$). The coefficient C_2 captures the fact that skipping operations are performed while merging the short lists, and therefore not all elements need to be pushed and popped from the heap. To estimate the running time of DivideSkip, we decide the coefficients C_1 and C_2 offline by running a subset of possible queries on the data set, collecting their merging times, and doing a linear regression. If filtering techniques are applied, we can decide C_1 and C_2 for each group separately to increase the accuracy of our estimation.

Estimating the Post-Processing Time: For each candidate from the T-occurrence problem, we need to compute the corresponding distance to the query to remove the false positives. This time can be estimated as

$$\text{(number of candidates)} \times \text{(average edit-distance time)}.$$

Therefore, the main problem becomes how to estimate the number of candidates after solving the T-occurrence problem. This problem has been studied in the literature recently. For instance, Mazeika et al. [87] studied how to estimate this number. The methodology in the SEPIA technique [61] in the context of selectivity estimation of approximate string queries could be adopted with minor modifications to solve our problem. The technique in [72] might also be applicable. While these techniques could be used in our context, they have two limitations. First, their estimation is not 100% accurate, and an inaccurate result could greatly affect the accuracy of the estimated post-processing time, thus affecting the quality of the selected nonhole lists. Second, this estimation may need to be done very often when choosing lists to discard, and therefore needs to be efficient.

We develop an efficient, *incremental* algorithm that can compute a *very accurate* number of candidates for query Q if list l_i is discarded. The algorithm is called ISC, which stands for "Incremental-Scan-Count," where its idea comes from an algorithm called ScanCount developed in [76]. Although the original ScanCount is not the most efficient one for the T-occurrence problem, it has the nice property that it can be run incrementally.

Figure 4.3 shows the intuition behind this ISC algorithm. First, we analyze the query Q on the original indexing structure without any lists discarded. For each string id in the collection, we remember how often it occurs on all the inverted lists of the grams in the query and store them in an array C. (Later we will discuss how to reduce the size of this data structure.) Now we want to know if a list is discarded, how it affects the number of occurrences of each

string id. For each string id r on list l belonging to gram g to be discarded, we decrease the corresponding value $C[r]$ in the array by the number of occurrences of g in the query string, since this string r will no longer have g as a nonhole gram.

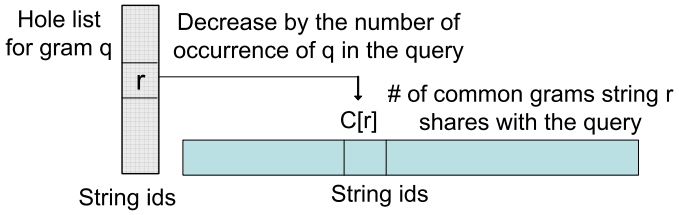


Figure 4.3: Intuition behind the Incremental-Scan-Count (ISC) algorithm.

After discarding this list for gram g, we first compute the new merging threshold T'. Then, we can find the new set of candidates by scanning the array C and recording those positions (corresponding to string ids) whose value is at least T'.

In the example in Figure 4.4 the hole list includes string ids 0, 2, 5, and 9. For each of the four ids, we decrease the corresponding value in the array by 1 (assuming the hole gram occurs once in the query). Suppose the new threshold T' is 3. We scan the new array to find those string ids whose number of occurrences among all non-hole lists is at least 3. These strings, which are 0, 1, and 9 (in bold face in the figure), are candidates for the query based on using the new threshold after this hole list is discarded.

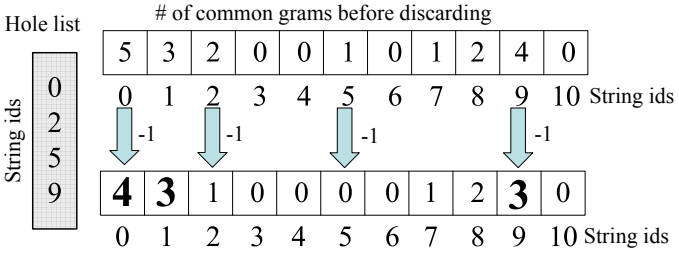


Figure 4.4: Running the ISC algorithm.

Since most of the values in the array are zero, we can keep a vector to store references to those positions in the array whose original values (without any lists discarded) are not zero.

Instead of scanning the whole array to find the new set of candidates we can scan this vector and follow the references to the elements in the array that may be candidates.

Estimating the Scan Time: If the new merging threshold $T' \leq 0$, then the query can only be answered by a scan. The time of a scan can be estimated in the same manner as the post-processing time, except that the number of candidates is the number of strings in the collection. If filtering techniques are applied then the number of candidates is the sum of all distinct string ids of groups that need to be considered by the query.

4.4.2 Algorithms for Choosing Inverted-Lists to Discard

In this section, we study how to wisely choose lists to discard in order to satisfy a given space constraint. The following are several simple approaches:

- LongList: Choose the longest lists to discard.
- ShortList: Choose the shortest lists to discard.
- RandomList: Choose random lists to discard.

These naive approaches blindly discard lists without considering the effects on query performance. Clearly, a good choice of lists to discard depends on the query workload. Based on our previous analysis, we present a cost-based algorithm called DiscardLists, as shown in Figure 4.5. Given the initial set of inverted lists, the algorithm iteratively selects lists to discard, based on the size of a list and its effect on the average query performance for a query workload $\mathcal{Q}$ if it is discarded. $\mathcal{Q}$ can be obtained from query logs, or from a uniformly distributed workload over the dataset when initially there is no query log available. The algorithm keeps selecting lists to discard until the total size of the remaining lists meets the given space constraint (line 2).

Algorithm: **DiscardLists**
Input: Inverted lists $L = \{l_1, \ldots, l_n\}$
Constraint B on the total list size
Query workload $\mathcal{Q} = \{Q_1, \ldots, Q_m\}$
Output: A set D of lists in L that are discarded
Method:

```
1.  D = ∅;
2.  WHILE (B < (total list size of L)) {
3.      FOR (each list l_i ∈ L) {
4.          Compute size reduction Δ^i_size if discarding l_i
5.          Compute difference of average query time Δ^i_time
                for queries in Q if discarding l_i
        }
6.      Use Δ^i_size's and Δ^i_time's of the lists to decide what
            lists to discard
7.      Add discarded lists to D
8.      Remove the discarded lists from L
    }
9.  RETURN D
```

Figure 4.5: Cost-based algorithm for choosing inverted lists to discard.

In each iteration (lines 3-8), the algorithm needs to evaluate the quality of each remaining list l_i, based on the expected effect of discarding this list. The effect includes the reduction Δ^i_{size} on the total index size, which is the length of this list. It also includes the change Δ^i_{time} on the average query time for the workload $\mathcal{Q}$ after discarding this list.[1]

In line 6 in Figure 4.5, in each iteration of the DiscardLists algorithm, we need to use the Δ^i_{size}'s and Δ^i_{time}'s of the lists to decide which lists should be really discarded. There are many different ways to make this decision. One way is to choose a list with the smallest Δ^i_{time} value (notice that it could be negative). Another way is to choose a list with the smallest $\Delta^i_{time}/\Delta^i_{space}$ ratio.

There are several ways to reduce the computation time of the estimation:

[1]Surprisingly, Δ^i_{time} can be both positive and negative, since in some cases discarding lists can even reduce the average running time for the queries.

(1) When discarding the list l_i, those queries whose strings do not have the gram of l_i will not be affected, since they will still have the same set of nonhole grams as before. Therefore, we only need to re-evaluate the performance of the queries whose strings have this gram of l_i. In order to find these strings efficiently, we build an inverted-list index structure for the *queries*, similar to the way we construct inverted lists for the strings in the collection. When discarding the list l_i, we can just consider those queries on the *query* inverted list of the gram for l_i.

(2) We run the algorithm on a random subset of the strings. As a consequence, (a) we can make sure the entire inverted lists of these sample strings can fit into a given amount of memory. (b) We can reduce the array size in the ISC algorithm, as well as its scan time to find candidates. (c) We can reduce the number of lists to consider initially since some infrequent grams may not appear in the sample strings.

(3) We run the algorithm on a random subset of the queries in the workload $\mathcal{Q}$, assuming this subset has the same distribution as the entire workload. As a consequence, we can reduce the computation to estimate the scan time, merging time, and post-processing time (using the ISC algorithm).

(4) We do not discard those very short lists, thus we can reduce the number of lists to consider initially.

(5) In each iteration of the algorithm, we choose multiple lists to discard based on the effect on the index size and overall query performance. In addition, for those lists that have very poor time effects (i.e., they affect the overall performance too negatively), we do not consider them in future iterations, i.e., we have decided to keep them in the index structure. In this way we can reduce the number of iterations significantly.

4.5 Compression Method 2: Combining Inverted Lists

In this section, we study how to reduce the size of an inverted-list index by *combining* some of the lists. Intuitively, when the lists of two grams are similar to each other, using a single inverted list to store the union of the two originial lists for both grams could save some space. One subtlety in this approach is that the string ids on a list are treated as a *set* of ordered elements (without duplicates), instead of a *bag* of elements. By combining two lists we mean taking the *union* of the two lists so that space can be saved. Notice that the T lower bound in the T-occurrence problem is derived from the perspective of the grams in the query. (See Equation 2.4 in Section 2.4 as an example.) Therefore, if a gram appears multiple times in a data string in the collection (with different positions), on the corresponding list of this gram the string id appears only once. If we want to use the positional filtering technique (mainly for the edit distance function) described in [44, 76], for each string id on the list of a gram, we can keep a range of the positions of this gram in the string, so that we can utilize this range to do filtering. When taking the union of two lists, we need to accordingly update the position range for each string id.

We will first discuss the data structure and the algorithm for efficiently combining lists in Section 4.5.1, and then analyze the effects of combining lists on query performance in Section 4.5.2. We also show that an index with combined inverted lists gives us a new opportunity to improve the performance of list-merging algorithms (Section 4.5.2.1). Based on our analysis we propose an algorithm called CombineLists for choosing lists to combine in the presence of space constraints while retaining efficient processing (Section 4.5.3).

4.5.1 Combining Lists

This section studies the data structure and algorithm used to combine lists. In the original inverted-list structure, each gram g is mapped to a list l of string ids: $g \rightarrow l$. Combining two lists l_1 and l_2 will produce a new list $l_{new} = l_1 \cup l_2$, which is also sorted by string ids. All grams that were previously mapped to l_1 and l_2 (there could be several grams due to earlier combining operations) will be mapped to l_{new}. In this fashion we can support combining more than two lists iteratively. At the first glance a reverse mapping from lists to grams may seem to be necessary, because we have to track all grams associated with a list in order to update them. However, a data structure called Disjoint-Set with the algorithm Union-Find [42] can be utilized to efficiently combine more than two lists. The Disjoint-Set data structure is a special forest in which each node holds only the pointer to its parent. To apply the Disjoint-Set structure to this problem, each inverted list corresponds to a tree and each node corresponds to a gram. All nodes in the same tree share the same inverted list, which is stored as a single copy attached to the root. Initially all grams are disjoint tree roots, with their individual inverted lists attached to each of them. As we combine two trees, the root of one of them becomes a child of the root of the other, which takes over all string ids from the list of the first tree to generate the union list. Figure 4.6 shows the Disjoint-Set structure used with the Union-Find algorithm when combining list of g_2 and list of g_3 while g_2 and g_1 are already sharing the same list $l_1 \cup l_2$. After the combination of the two trees corresponding to the two lists, the result list is $l_1 \cup l_2 \cup l_3$ and attached to root g_1. The Union-Find algorithm can also shorten the path from nodes to their root to speed up inverted-list lookup operations.

To satisfy a specific space constraint, we would like to know the exact size of saved space when combining lists. The size reduction of combining two lists l_1 and l_2 is computed as

$$\Delta_{size}^{(1,2)} = |l_1| + |l_2| - |l_1 \cup l_2| = |l_1 \cap l_2|. \tag{4.4}$$

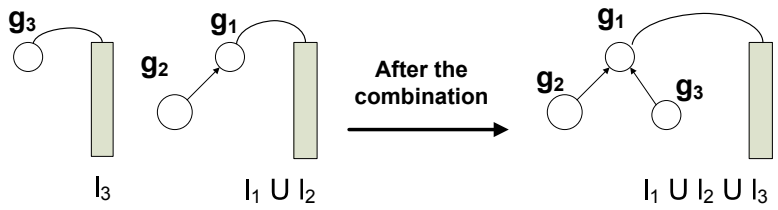


Figure 4.6: Combining list of g_2 with list of g_3 using Union-Find

The larger the intersection of two lists is, the more benefit we can get from combining them.

4.5.2 Effects of Combining Lists on Query Performance

In this section we study how query performance will be affected by combining lists. For a similarity query with a string s, if the lists of the grams in $G(s)$ are combined (possibly with lists of grams not in $G(s)$), then the performance of this query can be affected in the following ways:

- Different from the approach of discarding lists, the lower bound T in the T-occurrence problem remains the same, since an answer still needs to appear at least this number of times on the lists. Therefore, if a query was not in a panic case before, then it will not be in a panic case after combining inverted lists.
- The lists will become longer. As a consequence, it will take more time to traverse these lists to find candidates during list merging. In addition, more false positives may be produced to be post-processed.

4.5.2.1 Effects on List-Merging

As inverted lists get combined, some of them will become longer. Here we take one of the list-merging algorithms DivideSkip as an example. In Equation 4.3, as some $|l_i|$'s become larger, the time for merging the short lists and the time for verifying the candidates against

the long lists could increase. In this sense it appears that combining lists can only increase the list-merging time in query answering. However, the following observation opens up opportunities for us to further decrease the list-merging time given an index structure with combined lists.

We notice that a gram could appear in the query string s multiple times (with different positions), thus these grams share common lists. In the presence of combined lists, it becomes possible for even different grams in $G(s)$ to share lists. This sharing suggests a way to improve the performance of existing list-merging algorithms for solving the T-occurrence problem [104, 76]. A simple way to use one of these algorithms is to pass it a list for each gram in $G(s)$. Thus we pass $|G(s)|$ lists to the algorithm to find string ids that appear at least T times on these (possibly shared) lists. We can improve the performance of the algorithm as follows. We first identify the shared lists for the grams in $G(s)$. For each *distinct* list l_i, we also pass to the algorithm the number of grams sharing this list, denoted by w_i. Correspondingly, the algorithm needs to consider these w_i values when counting string occurrences. In particular, if a string id appears on the list l_i, the number of occurrences should increase by w_i, instead of "1" in the traditional setting. In this way, we can reduce the number of lists passed to the algorithm, thus possibly even reducing its running time. The algorithms in [104] already consider different list weights, and the algorithms in [76] can be modified slightly to consider these weights.[2]

4.5.2.2 Effects on Post-processing

After combining two lists l_1 and l_2, more candidates will be generated for post-processing. The problem becomes computing the number of candidates generated from the list-merging algorithm. We notice that although different list-merging algorithms can have different

[2]Interestingly, our experiments showed that, even for the case we do not combine lists, this optimization can already reduce the running time of existing list-merging algorithms by up to 20% for a data set with duplicate grams in a string.

performances, the results of the T-occurrence problem are exactly the same given the same index structure. Therefore we can use any list-merging algorithm to compute the number of candidates. Before combining any lists, the candidate set generated from a list-merging algorithm contains all correct answers and some false positives. We are particularly interested to know how many new false positives will be generated by combining two lists l_1 and l_2. The ISC algorithm described in Section 4.4.1.4 can be modified to adapt to this setting.

Recall that in the ISC algorithm, a ScanCount vector is maintained for a query Q to keep track of the number of grams Q shares with each string id in the data set. The strings whose corresponding values in the ScanCount vector are at least T will be candidate answers. By combining two lists l_1 and l_2, the lists of those grams that are mapped to l_1 or l_2 will be conceptually extended. Figure 4.7 describes the intuition of the algorithm. Every gram that was previously mapped to l_1 or l_2 will now be mapped to $l_1 \cup l_2$. The extended part of l_1 is $ext(l_1) = l_2 \backslash l_1$, where "\" denotes the difference of the list-element sets. Let $w(Q, l_1)$ denote the number of times grams of Q reference l_1. The ScanCount value of each string id in $ext(l_1)$ will be increased by $w(Q, l_1)$. Since for each reference, all string ids in $ext(l_1)$ should have their ScanCount value increased by one, the total incrementation will be $w(Q, l_1)$ (*not* $w(Q, l_2)$). The same operation needs to be done for $ext(l_2)$ symmetrically. It is easy to see that the ScanCount values are monotonically increasing as lists are combined. The strings whose ScanCount values increase from below T to at least T become new false positives after l_1 and l_2 are combined.

Figure 4.8 shows an example in which $l_1 = \{0, 2, 8, 9\}$ and $l_2 = \{0, 2, 3, 5, 8\}$. Before combining l_1 and l_2, two grams of Q are mapped to l_1 and three grams are mapped to l_2. Therefore, $w(Q, l_1) = 2$ and $w(Q, l_2) = 3$. For every string id in $ext(l_1) = \{3, 5\}$, their corresponding values in the ScanCount vector will be increased by $w(Q, l_1)$. Let C denote the ScanCount vector. $C[3]$ will be increased from 6 to 8, while $C[5]$ will be increased from 4 to 6. Given the merging threshold $T = 6$, the change on C[5] indicates that string 5 will become a new

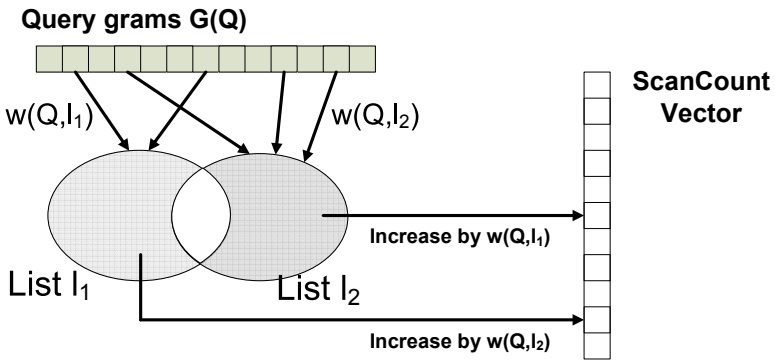


Figure 4.7: ISC algorithm for computing the number of candidates. We compute the new number of candidates for query Q after combining lists l_1 and l_2 by incrementing the appropriate positions in the ScanCount Vector.

false positive. The same operation is carried out on $ext(l_2)$.

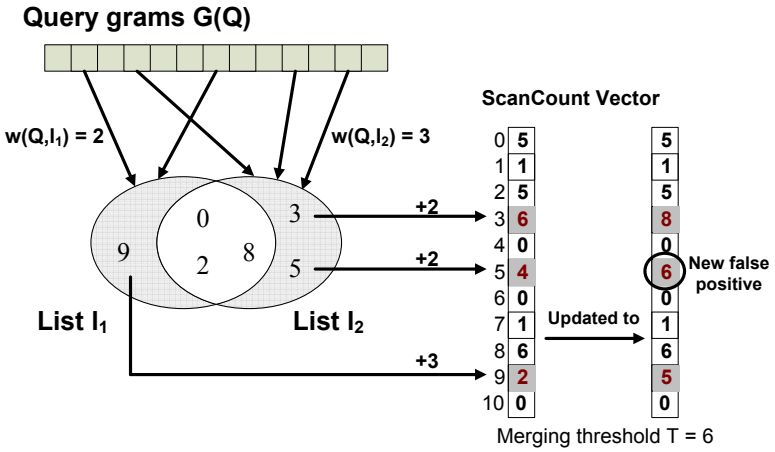


Figure 4.8: Example of ISC algorithm after combining lists l_1 and l_2.

4.5.3 Algorithms for Choosing Lists to Combine

Based on the previous analysis, we present algorithms for choosing inverted lists to combine.

4.5.3.1 Basic algorithm

The basic algorithm consists of two steps: discovering candidate gram pairs that are correlated, and selecting some of them to combine.

Step 1: Discovering Candidate Gram Pairs We are interested in combining *correlated* lists. Jaccard similarity is used to measure the correlation of two lists. It is defined as:

$$Corr(l_1, l_2) = \frac{|l_1 \cap l_2|}{|l_1 \cup l_2|}. \tag{4.5}$$

Two lists are considered for combining only if their correlation is greater than a threshold. Clearly it is computationally prohibitive to consider all pairs of grams. Here we present two efficient methods for generating such pairs.

- *Using Adjacent Grams*: We only consider pairs of *adjacent* grams in the strings. If we use q-grams to construct the inverted lists, we can just consider those $(q+1)$-grams. Each such gram corresponds to a pair of q-grams. For instance, if $q = 3$, then the 4-gram `tion` corresponds to the pair (`tio`, `ion`). For each such adjacent pair, we treat it as a candidate pair if the Jaccard similarity of their corresponding lists (Equation 4.5) is greater than a predefined threshold.
- *Using Locality-Sensitive Hashing*: The above approach cannot find strongly correlated grams that are not adjacent in strings. In the literature there are efficient techniques for finding strongly correlated pairs of lists. One of them is called Locality-Sensitive Hashing (LSH) [56]. Using a small number of so-called MinHash signatures for each list, we can use LSH to find those gram pairs whose lists satisfy the above correlation condition with a high probability.

Step 2: Selecting Candidate Pairs to Combine

The second step is selecting candidate pairs to combine. We iteratively pick gram pairs and combine their lists if their correlation satisfies the threshold. Notice that each time we process a new candidate gram pair, since the list of each of them could have been combined with other lists, we still need to verify their (possibly new) correlation before deciding whether we should combine them. After processing all these pairs, we check if the index size meets a given space constraint. If so, the process stops. Otherwise, we decrease the correlation threshold and repeat the process above until the new index size meets the given space constraint.

This basic algorithm blindly chooses candidate pairs to combine without considering the consequence on the overall query performance. Based on our previous discussions, we propose a new algorithm to wisely choose lists to combine in the second step.

4.5.3.2 Cost-Based Algorithm

Figure 4.9 shows the cost-based algorithm, which takes the estimated cost of a query workload into consideration when choosing lists to combine. It iteratively selects pairs to combine, based on the space saving and the impact on the average query performance of a query workload $\mathcal{Q}$. The algorithm keeps selecting pairs to combine until the total size of the inverted lists meets a given space constraint B. For each gram pair (g_i, g_j), we need to get their current corresponding lists, since their lists could have been combined with other lists (lines 3 and 4). We check whether these two list pointers are the same (line 5), and also whether their correlation is above the threshold (line 6). Then we compute the size reduction (line 8) and estimate the average query time difference based on Equation 4.3 and the ISC algorithm (line 9), based on which we decide the next list pair to combine (lines 10 and 11).

We can use similar optimization techniques as described in 4.4 to improve the performance of CombineLists.

Algorithm: **CombineLists**
Input: Candidate gram pairs $P = \{(g_i, g_j)\}$
Constraint B on the total list size
Query workload $\mathcal{Q} = \{Q_1, \ldots, Q_m\}$
Output: Combined lists.
Method:

```
1.  WHILE ((expected total index size) > B ) {
2.      FOR (each gram pair (g_i, g_j) ∈ P) {
3.          l_i = current list of g_i
4.          l_j = current list of g_j
5.          if (l_i and l_j are the same list as reference
6.              or corr(l_i, l_j) < δ)
7.              remove (g_i, g_j) from P and continue
8.          Compute size reduction Δ_size^(l_i,l_j) if combining l_i, l_j
9.          Compute difference of average query time Δ_time^(l_i,l_j)
                for queries in Q if combining l_i, l_j
        }
10.     Use Δ_size^(l_i,l_j)'s and Δ_time^(l_i,l_j)'s of the gram pairs to decide
            which pair to combine
11.     Combine the two lists l_i and l_j based on the decision
12.     Remove the combined gram pair from P
    }
```

Figure 4.9: CombineLists algorithm to select gram pairs to combine.

4.6 Experiments

In this section, we report experimental results for the proposed techniques on the following three real data sets.

- **IMDB Actor Names**: It consists of the actor names downloaded from the IMDB website[3]. There were 1,199,299 names. The average string length was 17 characters.
- **WEB Corpus Word Grams**: This data set[4], contributed by Google Inc., contained word grams and their observed frequency counts on the Web. We randomly chose 2 million records with a size of 48.3MB. The number of words of a string varied from 3 to 5. The average string length was 24.

[3] www.imdb.com
[4] www.ldc.upenn.edu/Catalog, number LDC2006T13

- **DBLP Paper Titles**: It includes paper titles downloaded from the DBLP Bibliography site[5]. It had 274,788 paper titles. The average string length was 65.

For all experiments the gram length q was 3 and the inverted-list index was held in main memory. Also, for the cost-based DiscardLists and CombineLists approaches, by doing sampling we guaranteed that the index structures of the sample strings fit into memory. We used the DivideSkip algorithm described in [76] to solve the T-occurrence problem due to its high efficiency. From each data set we used 1 million strings to construct the inverted-list index (unless specified otherwise). We tested query workloads using different distributions, e.g., a Zipfian distribution or a uniform distribution. To do so, we randomly selected 1,000 strings from each data set and generated a workload of 10,000 queries according to some distribution. We conducted experiments using edit distance, Jaccard similarity, and cosine similarity. We mainly focused on the results of edit distance (with a threshold of 2). We report additional results for other functions in Section 4.7.2. All of the algorithms were implemented using GNU C++ and run on a Dell PC with 2GB main memory and a 3.4GHz Dual Core CPU running the Ubuntu 8.04 operating system.

4.6.1 Evaluating the Carryover-12 Compression Technique

We first evaluated the performance of the Carryover-12-based compression technique discussed in Section 4.3. We used a segment size of 128 4-byte integers, which was a good value for the performance and compression ratio, and examined the query performance with different cache sizes. Figure 4.10 shows that query performance benefits from using a larger cache. On the WebCorpus data set, when we used no cache the average query time was 64.4ms, which is more than 8 times the average query time with a cache of 5,000 slots. Since the whole purpose of compressing an inverted lists is to save space, it is contradictory to improve query performance by increasing the cache size indefinitely. Figure 4.10 shows that 20,000 is

[5] www.informatik.uni-trier.de/~ley/db

a reasonable number of cache slots (approximately 10MB), so we used this number of slots for experiments with Carryover-12 throughout this section.

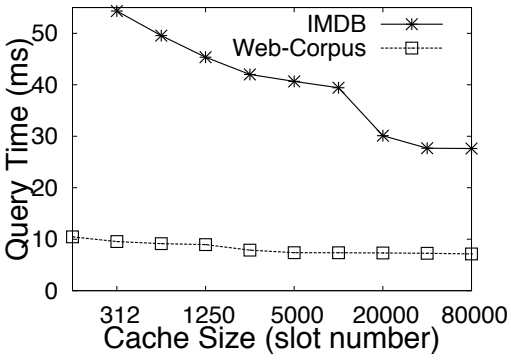


Figure 4.10: Benefits of using cache when decoding compressed lists.

4.6.2 Evaluating the DiscardLists Algorithm

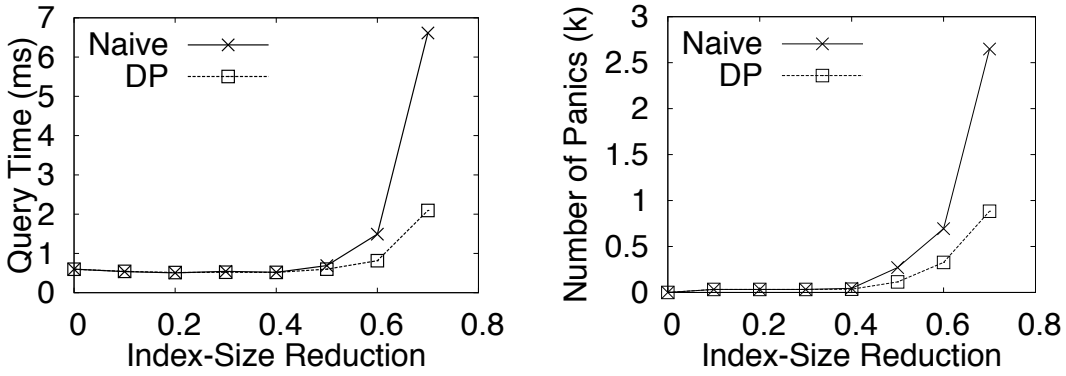
In this section we evaluate the performance of the DiscardLists algorithm for choosing inverted-lists to discard. In addition to the three basic methods to choose lists to discard (LongList, ShortList, and RandomList), we also implemented the following cost-based methods:

- PanicCost: In each iteration we discard the list with the smallest ratio between the list size and the number of additional panic cases. Another similar approach, called PanicCost$^+$, simply discards the list with the smallest number of additional panic cases, disregarding the list length.
- TimeCost: It is similar to PanicCost, except that we use the ratio between the list size and the total time effect of discarding a list (instead of the number of additional panics). Similarly, an approach called TimeCost$^+$ discards the list with the smallest time effect.

The index-construction time mainly consisted of two major parts: selecting lists to discard and generating the final inverted-list structure. The time for generating samples was negligible. For the LongList, ShortList, and Random approaches, the time for selecting lists to

discard was small, whereas in the cost-based approaches the list-selection time was prevalent. In general, increasing the size-reduction ratio also increased the list-selection time. For instance, for the IMDB dataset, at a 70% reduction ratio, the total index-construction time for the simple methods was about half a minute. The construction time for PanicCost and PanicCost$^+$ was similar. The more complex TimeCost and TimeCost$^+$ methods needed 108s and 353s, respectively.

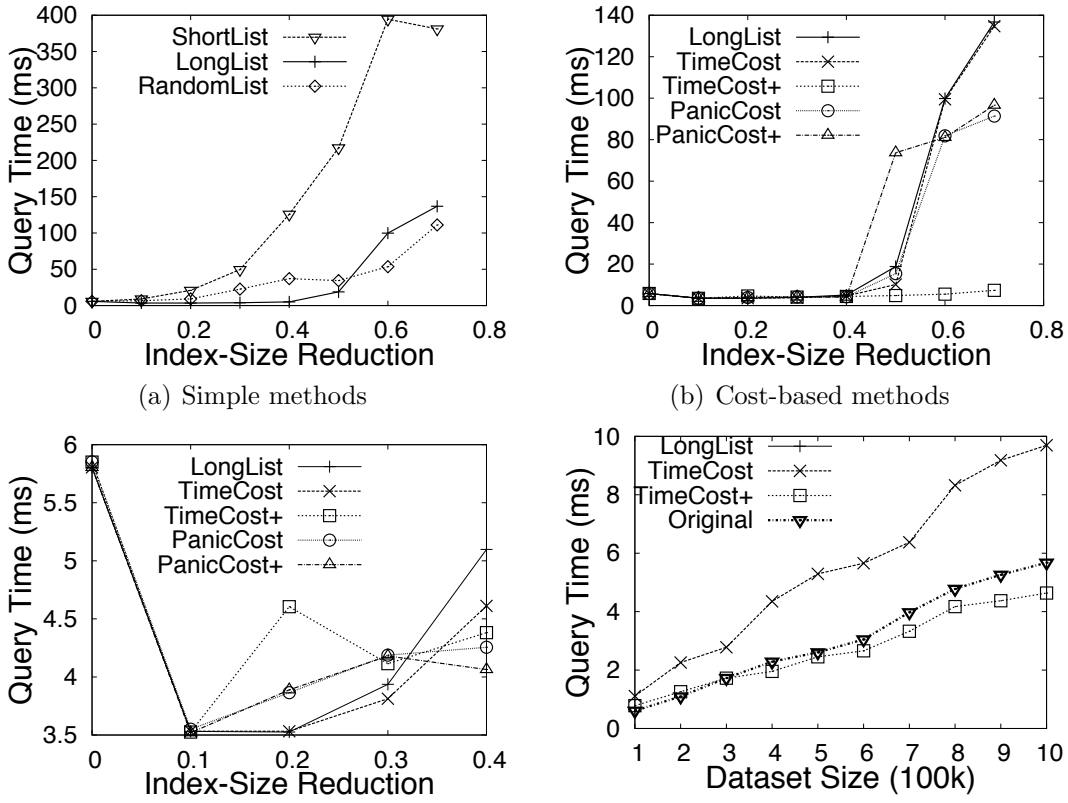
Benefits of Tighter Bounds: We first examined the effects of using the dynamic programming algorithm for computing a tighter bound in the presence of hole lists, as described in Section 4.4.1.1. We used the DBLP data set, and ran a workload of 10,000 queries with a Zipfian distribution. We used the LongList method to reduce the index size. Figure 4.11(a) shows the average running time for both the naive method for computing a bound using Equation 4.1 (marked as "Naive"in the figure) and the dynamic programming algorithm for computing a bound (marked as "DP"). The x-axis is the total memory reduction ratio, where "0" means no size reduction, while the y-axis is the average query time. We see that the dynamic programming approach indeed computed a tighter bound, which notably reduced the average query time. This benefit is largely due to the decrease in the number of panics, as illustrated in Figure 4.11(b). In the interest of brevity we omit the results for the other data sets since they showed a very similar trend.



(a) Average Query Time (b) Number of Panics

Figure 4.11: Benefits of using dynamic programming to tighten bounds.

Comparing Simple Methods with Cost-Based Methods: Next we compare the simple methods for choosing lists to discard with the cost-based ones. For each data set, we conducted experiments by indexing one million strings from the IMDB and WebCorpus datasets and running a workload of 10,000 queries with a Zipfian distribution (for brevity we do not show the results of the DBLP data set, which were consistent with the results of the other two data sets). For those cost-based approaches, we used a sampling ratio of 0.1% for the data strings and a ratio of 25% for the queries.



(a) Simple methods

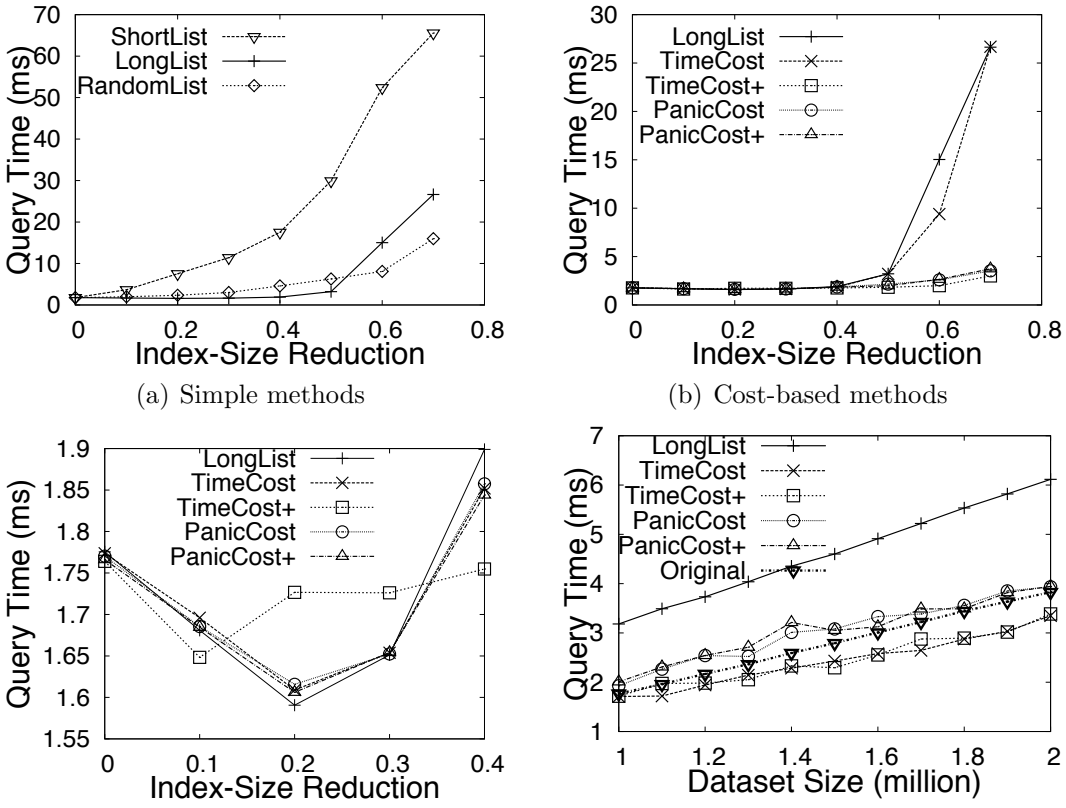
(b) Cost-based methods

(c) Cost-based methods (zoomed in)

(d) Scalability

Figure 4.12: Reducing index size by discarding lists (IMDB Actors).

We first considered three simple methods, namely LongList, ShortList, and Random. Figures 4.12(a) and 4.13(a) show how discarding lists affected the average query performance as we increased the index-size-reduction ratio. We can see that the LongList and Random methods were equally promising, providing compression ratios up to 50% with little penalty



(a) Simple methods

(b) Cost-based methods

(c) Cost-based methods (zoomed in)

(d) Scalability

Figure 4.13: Reducing index size by discarding lists (WebCorpus word grams).

in the query execution time. For instance, in Figure 4.12(a), when there was no index reduction, the original average query time was 5.8ms. When the reduction ratio was 50%, the running time increased to 18.8ms for the LongList method. For all the data sets, the ShortList method performed significantly worse than the other two. The main reason is that it caused a dramatic increase in the merge-time and post-processing time. For higher reduction ratios, RandomList sometimes outperformed LongList because the former avoided more panics alleviated in LongList by discarding lists of popular grams.

Based on these results, we chose to omit ShortList and Random in the following experiments. Figures 4.12(b) and 4.13(b) show the benefits of employing the cost-based methods to select lists to discard. Most noteworthy of which is the TimeCost$^+$ method, which consistently delivered the best query performance. As shown in Figure 4.12(b), the method achieved a

70% reduction ratio while increasing the query processing time from the original 5.8ms to 7.4ms only. All the other methods increased the time up to at least 96ms for that reduction ratio. Notice that TimeCost$^+$ ignored the list size when selecting a list to discard, so the good results may seem counter-intuitive. A deeper analysis suggests that there is really no obvious need to take the list size into account when choosing lists to discard because discarding one long lists is not necessarily better than discarding many small ones amounting to the same space savings. The reason for this behavior is that discarding lists may improve or degrade query performance, so it is not necessarily better to terminate the list-discarding algorithm as early as possible by preferring longer lists. Thus, our main objective should be minimizing the penalty in query performance, regardless of list sizes. We see that the TimeCost method behaved similarly to LongList. The reason is that the former preferred discarding long lists over short lists because it tries to maximize the ratio of the list-size and the estimated running time of the query workload. The PanicCost and PanicCost$^+$ methods performed well on the DBLP and WebCorpus data sets because at higher reduction ratios the cost for panics became prevalent. The two panic-based methods both performed poorly in Figure 4.12(b) because they blindly minimized the number of panics while drastically increasing postprocessing time. TimeCost$^+$ beat all the other methods because it can balance the merging time, post-processing time, and scan time.

Surprising Improvement on Performance: Figures 4.12(c) and 4.13(c) show more details from the previous figure where the reduction ratio is smaller (less than 40%). A surprising finding is that, even for low to moderate reduction ratios, discarding lists could improve the query performance! Take Figure 4.12(c) for an example. All the methods reduced the average query time from the original 5.8ms to 3.5ms for a 10% reduction ratio. The main reason for this performance improvement is that by discarding long lists we can help the list-merging algorithms solve the T-occurrence problem more efficiently. We see that significantly reducing the number of total list-elements to process can more than compensate for the decrease in the merging threshold.

Scalability: To test scalability, for each data set, we increased its number of strings and used 50% as the index-size-reduction ratio. Figures 4.12(d) and 4.13(d) show that the TimeCost$^+$ method performed consistently well, even outperforming the corresponding uncompressed indexes (indicated by "original"). As the data size increased, TimeCost$^+$ began outperforming the uncompressed index. For example, at 1 million strings the average query time decreased from 5.7ms (original) to 4.6ms (TimeCost$^+$). This benefit can be attributed to the skewed Zipfian query distribution that this algorithm adapts to. Notice that the TimeCost method behaves predictably, albeit badly, whereas the two panic-based methods can behave unpredictably because the merge operation becomes increasingly expensive with a growing data set. Therefore, we excluded the curves for PanicCost and PanicCost$^+$ from 4.13(d) since they deliver no clear trend.

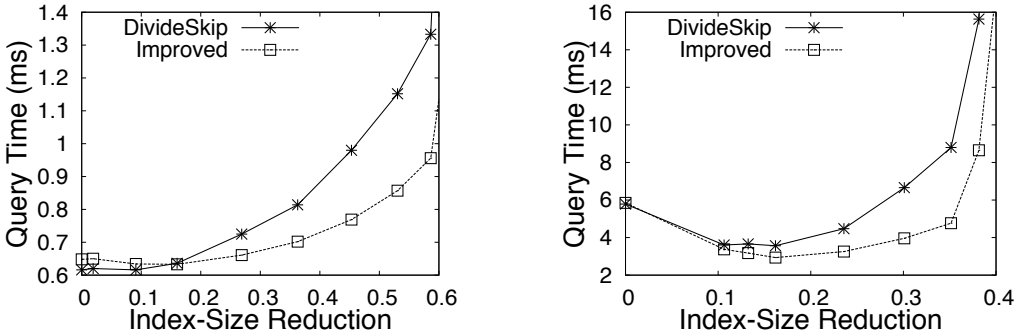
Summary: (1) Discarding lists can achieve a significant index size reduction without significantly increasing query running time. (2) Even for small reduction ratios, we can even improve query performance. (3) For most cases the TimeCost$^+$ method achieves a good query performance, making it the algorithm of choice here.

4.6.3 Evaluating the CombineLists Algorithm

We next evaluated the performance of the CombineLists algorithm on the same three data sets. In step 1, we implemented three methods to generate candidate list pairs: (1) Adjacent: Consider $(q + 1)$-grams. (2) LSH: Use LSH to find strongly correlated pairs (with a high probability). For this purpose, we generated 100 MinHash signatures for each list. (3) Adjacent+LSH: We take the union of the candidate pairs generated by these two methods. In step 2, we implemented both the CombineBasic and the CombineCost algorithms for iteratively selecting list pairs to combine.

Benefits of Improved List-Merging Algorithms: We first evaluated the benefits of

using the improved list-merging algorithms to solve the T-occurrence problem for queries on combined inverted lists, as described in Section 4.5.2. As an example, we compared the DivideSkip algorithm in [76] and its improved version that considers duplicated inverted lists in a query. We used the Adjacent+LSH method to generate candidate list pairs and the CombineBasic algorithm to select lists to combine. Figure 4.14 shows the average running time for the basic DivideSkip (marked as "DivideSkip") and the improved DivideSkip algorithm (marked as "Improved"). We can see that when the reduction ratio increased, more lists were combined, and the improved algorithm did reduce the average query time.

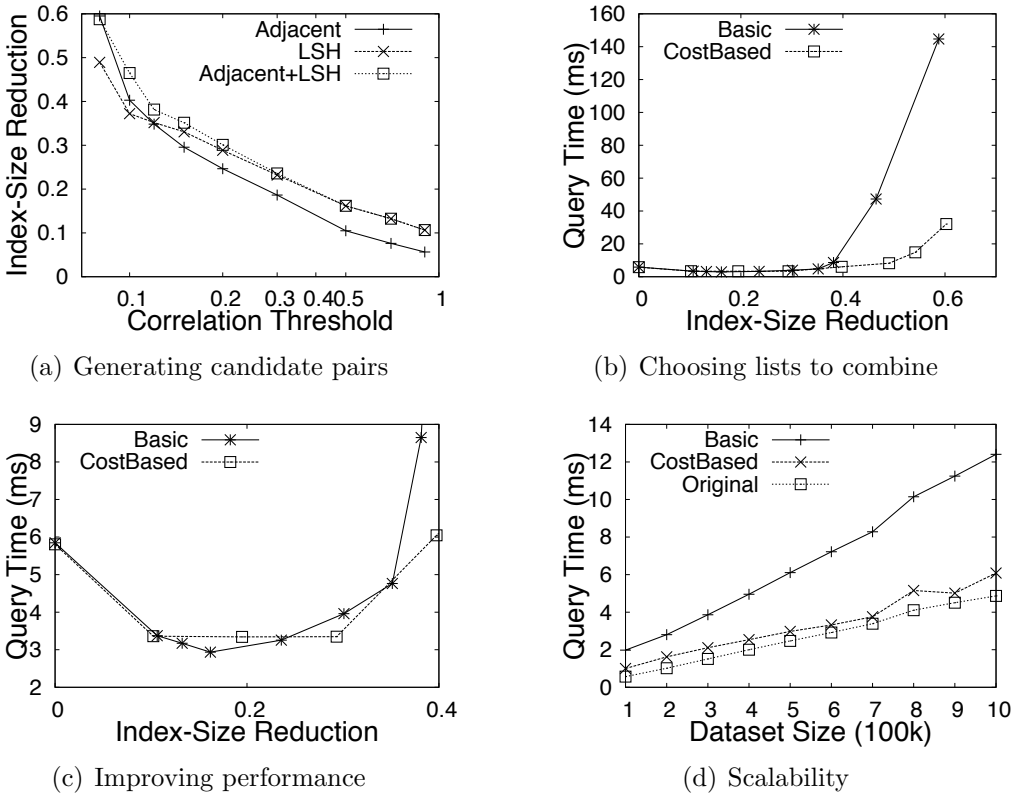


(a) DBLP titles (b) IMDB actors

Figure 4.14: Reducing query time using improved list-merging algorithm (DivideSkip).

Generating Candidate List Pairs: Figures 4.15(a) and 4.16(a) compare the index size reduction ratio for different candidate-generating methods, over various correlation-threshold values δ, on the two data sets. We used the CombineBasic method for combining lists. The reduction ratio was plotted in a logarithmic scale. We observed that on the IMDB and DBLP data sets, with a higher correlation threshold, LSH produced more candidate pairs than Adjacent since the former can find correlated pairs of grams that are not adjacent. The results also suggest that a significant number of correlated pairs were indeed from adjacent grams. As we decreased the correlation threshold, LSH was less capable of discovering all qualified candidate pairs without using more MinHash signatures. The results of the Adjacent+LSH approach were better than the two previous methods since it considered more

candidate list pairs.



(a) Generating candidate pairs

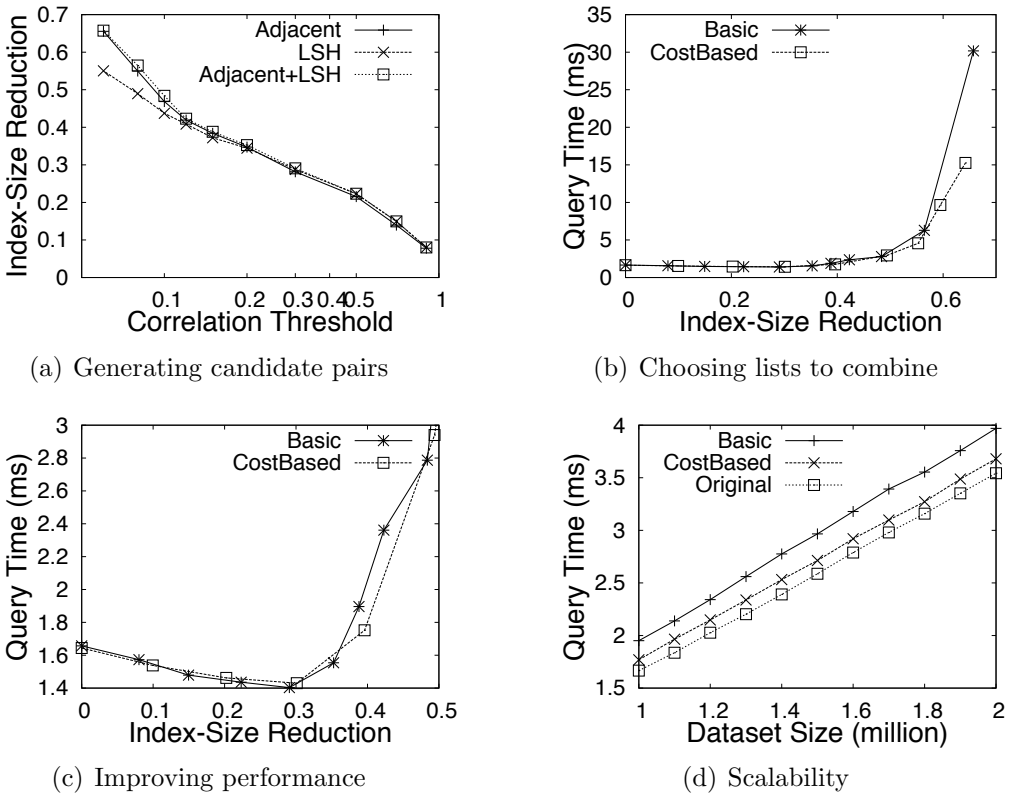
(b) Choosing lists to combine

(c) Improving performance

(d) Scalability

Figure 4.15: Reducing index size by combining lists (IMDB Actors).

Choosing Lists to Combine: We compared the CombineBasic algorithm with the cost-based CombineCost algorithm for choosing lists to combine, and the results are shown in Figure 4.16(b) and 4.15(b). The average query time was plotted over different reduction ratios for both algorithms. We observe that on all three data sets, the query running time for both algorithms increased very slowly as we increased the index size reduction ratio, until about 40% to 50%. That means that this technique can reduce the index size without increasing the query time! As we further increased the index size reduction, the query time started to increase. For the CombineCost algorithm, the time increased slowly, especially on the IMDB data set. The reason is that this cost-based algorithm avoided choosing bad lists to combine, while the CombineBasic algorithm blindly chose lists to combine. The difference between these two algorithms on the IMDB data set was larger than the other two data sets



(a) Generating candidate pairs

(b) Choosing lists to combine

(c) Improving performance

(d) Scalability

Figure 4.16: Reducing index size by combining lists (WebCorpus word grams).

because the CombineBasic algorithm required queries to spend a lot of post-processing time on this data set.

Figures 4.15(c) and 4.16(c) show that even when the reduction ratio is less than 40%, the query time decreased. This improvement is mainly due to the improved list-merging algorithms. Figures 4.15(d) and 4.16(d) show how the algorithms for combining lists affected query performance as we increased the data size for a reduction ratio of 40%.

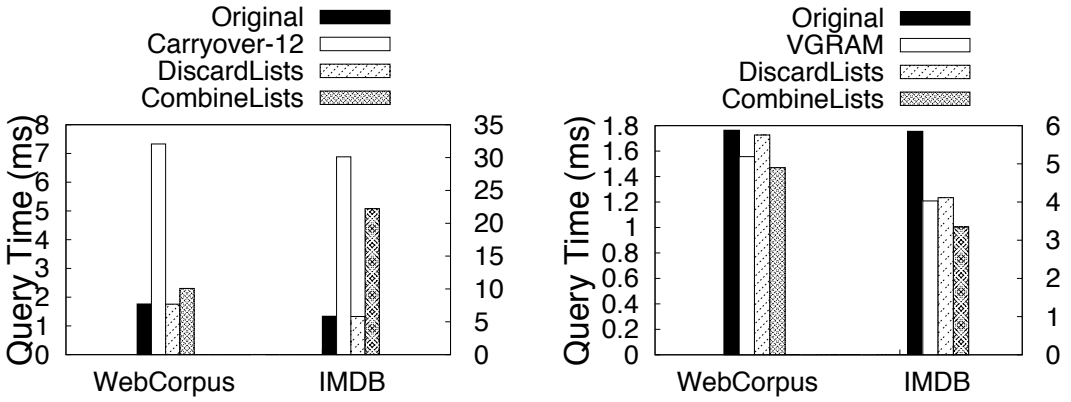
Summary: (1) Combining lists can also achieve a significant index size reduction without increasing query running time much. (2) For small reduction ratios, combining lists can also improve query performance, with the cause attributed to the improved list-merging algorithms. (3) For most cases the cost-based CombineCost algorithm gives a better index structure than the CombineBasic algorithm.

4.6.4 Comparing Different Compression Techniques

We implemented the compression techniques discussed so far as well as the VGRAM technique. Since Carryover-12 and VGRAM do not allow explicit control of the compression ratio, for each of them we reduced the size of the inverted-list index and computed their compression ratio. Then we compressed the index using DiscardLists and CombineLists separately to achieve the same compression ratio.

Comparison with Carryover-12: Figure 4.17(a) compares the performance of the two new techniques with Carryover-12. For Carryover-12, to achieve a good balance between the query performance and the index size, we used fixed-size segments of 128 4-byte integers and a synchronization point for each segment. The cache contained 20,000 segment slots (approximately 10MB). It achieved a compression ratio of 58% for the IMDB dataset and 48% for the WebCorpus dataset. We see that its online decompression has an impact on the performance. It increased the average running time from an original 5.85ms to 30.1ms for the IMDB dataset, and from an original 1.76ms to 7.32ms for the WebCorpus dataset. The CombineLists method performed significantly better at 22.15ms for the IMDB dataset and 2.3ms for the WebCorpus dataset. The DiscardLists method could even slightly decrease the running time compared to the original index, moving it to 5.81ms and 1.75ms for the IMDB and WebCorpus datasets, respectively.

Comparison with VGRAM: Figure 4.17(b) compares the performance of two new techniques with VGRAM. We set its q_{min} parameter to 4. We did not take into account the memory requirement for the dictionary trie structure because it was negligible. The compression ratio was 30% for the IMDB dataset and 27% for the WebCorpus dataset. Interestingly, all methods could outperform the original, uncompressed index. As suspected, VGRAM can considerably reduce the running time for both datasets. For the IMDB dataset, it reduced the time from an original 5.85ms to 4.02ms, and for the WebCorpus dataset from 1.76ms

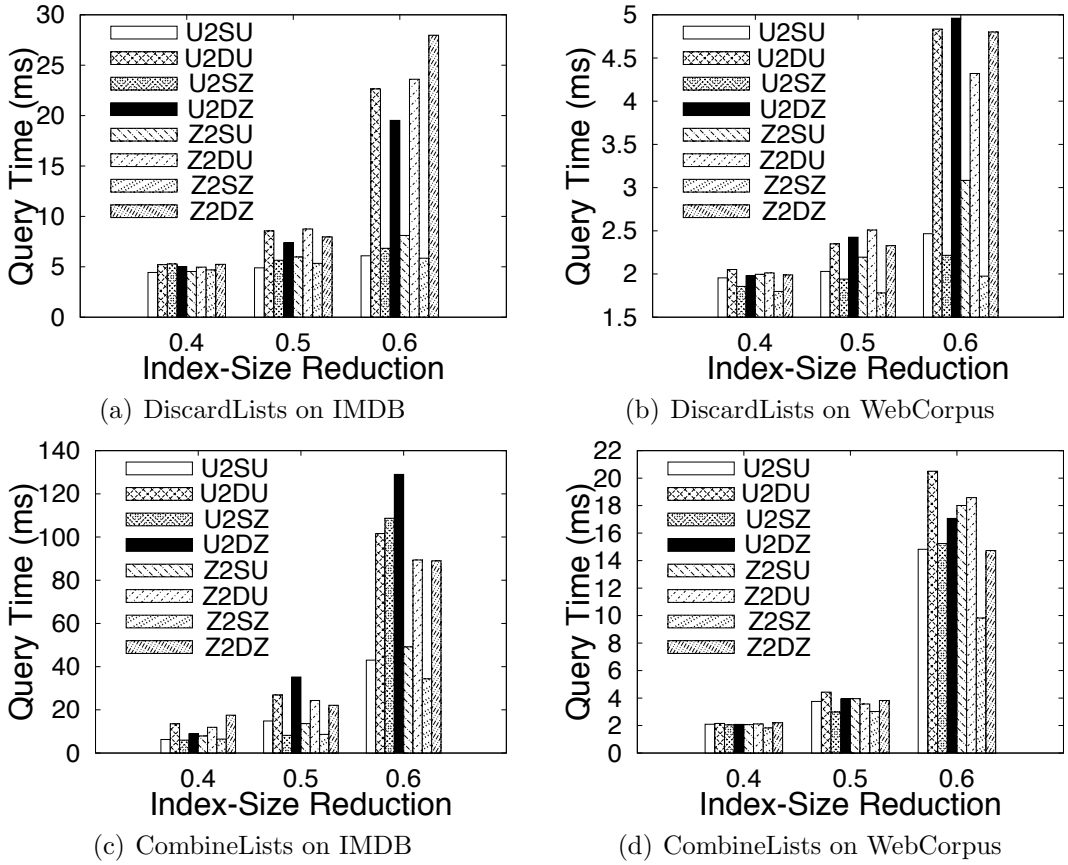


(a) Carryover-12 (b) VGRAM

Figure 4.17: Comparing DiscardLists and CombineLists with existing techniques. For a fair comparison, we used the same redution ration in all techniques. In each figure, the left scale corresponds to the WebCorpus data set, and the right scale corresponds to the IMDB data set.

1.55ms. Perhaps surprisingly, the CombineLists method reduced the running time even more than VGRAM to 3.34ms for the IMDB dataset and to 1.47ms for the WebCorpus dataset. The DiscardLists method performed competitively for the IMDB dataset at 3.93ms and slightly faster than the original index (1.67ms) on the WebCorpus dataset.

Summary: (1) CombineLists and DiscardLists can significantly outperform Carryover-12 at the same memory reduction ratio because of the costly online decompression required by Carryover-12. (2) For small compression ratios CombineLists performs best, even outperforming VGRAM. (3) For large compression ratios DiscardLists delivers the best query performance. (4) While Carryover-12 can achieve reductions up to 60% and VGRAM up to 30%, neither allows explicit control over the reduction ratio. DiscardLists and CombineLists offer this flexibility with good query performance.



(a) DiscardLists on IMDB

(b) DiscardLists on WebCorpus

(c) CombineLists on IMDB

(d) CombineLists on WebCorpus

Figure 4.18: Performance of DiscardLists and CombineLists on changing workloads.

4.6.5 Changing Query Workloads

The cost-based methods for discarding and combining lists assume a known query workload. We experimentally evaluated their performance in the presence of changing query workloads. That is, we build an inverted-list index assuming one query workload and measured the performance of another query workload. We shall denote the set of queries used to build the index as the *training workload* and the set of queries whose time we measured as the *testing workload*. For the testing workload, we consider that it can differ from the training workload both in the distinct query strings, and in the distribution of the distinct queries in the workload. To explain this further, let us examine the legends in Figure 4.18. The abbreviation scheme reflects (a) the distribution of the training workload, U for uniform and

Z for Zipfian, (b) whether the running workload consisted of the same distinct queries (S) as the training workload or entirely different distinct queries (D), and (c) the distribution of the running workload (the last U or Z). For example "U2SZ" means we used a uniformly distributed query workload as a training workload, and we generated a testing workload using the same distinct queries as in the training workload, but in a Zipfian distribution. Consider another example, "U2DZ". This means our training workload was a uniformly distributed query workload and the testing workload was generated from entirely different distinct queries to form a Zipfian distribution. Our query workloads consisted of 10,000 queries generated from 1,000 distinct queries. Figures 4.18(a) and 4.18(b) depict the average query performance of various workloads on indexes compressed by discarding lists. Not surprisingly, as the reduction ratio increases so does the variance in performance among the different workloads. For example, on the IMDB dataset at a 40% reduction ratio, Z2SZ took an average of 4.69ms per query versus a 4.97ms for Z2DU, while at a 60% reduction ratio the former increased to 5.85ms and the latter to 23.6ms. Interestingly, the experiments show that a change in both the distribution and the queries does not always present the worst case for performance. Consider the IMDB dataset at a 60% reduction ratio where one might suspect that U2DZ would perform worse than U2DU. The former yielded an average of 7.38ms and the latter 8.56ms. Notice that for both the IMDB and the WebCorpus datasets the worst performance achieved by changing the workloads is still competitive to a Carryover-12 compressed index. Take the WebCorpus dataset, where at a 50% reduction, Z2DU presents the worst change, to 2.5ms, while Carryover-12 needs an average of 7.32ms at a 48% reduction ratio.

Combining Lists: Figures 4.18(c) and 4.18(d) show the average query performance of various workloads on indexes compressed by combining lists. Again, the variance in performance increases as the compression ratio increases. For example on the WebCorpus dataset, at a 40% reduction ratio U2SU needed 2.09ms per query and U2DZ 2.08ms, while at a 60% ratio the former increased to 14.82ms and the latter to 17.06ms. Overall, combining lists does

not seem to be as sensitive to changes in the workload as discarding lists. This is because discarding lists can cause panic cases (requiring a scan) while combining lists cannot. For the WebCorpus dataset at a 50% reduction ratio, the worst change in workload (U2DU) amounted to 4.43ms per query, while at a 48% reduction ratio Carryover-12 required 7.32ms.

Summary: (1) Not surprisingly, the cost-based methods of combining and discarding lists are sensitive to how representative the chosen workload is. (2) The method of combining lists is less sensitive to changes in workloads than discarding lists, but it performs worse than discarding lists at high compression ratios. (3) Even for the worst changes in workloads that we tested, our techniques can mostly outperform Carryover-12 at similar compression ratios.

4.7 Other Issues

4.7.1 Index-Size Reduction with Filters

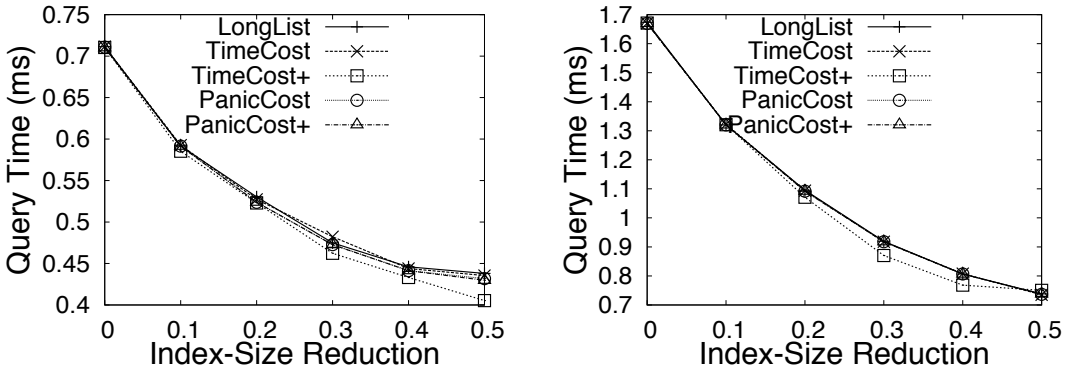
Various filtering techniques have been proposed to prune strings that cannot be similar enough to a query string [44, 28, 76]. In general, a filter can partition strings into different disjoint groups. There are different ways to adopt our size-reduction techniques in the presence of filters. Let $P_1, \ldots, P_n$ be the groups of the strings generated by filters. Two possible ways are, (1) *Global reduction*: Use the lists of the entire data set to decide what lists to discard or combine, then apply these decisions to the gram lists within each group P_i. (2) *Local reduction*: First decide how much reduction to allocate to each group P_i in order to achieve a global reduction ratio. Within each group G_i, decide the lists to discard or combine, allowing different groups to have different decisions. Our cost-based size-reduction methods can benefit from this divide-and-conquer strategy in two ways: (a) Running the DiscardLists or CombineLists algorithm for each group can be faster than running them once on the entire data set. (b) The performance improvement can allow a higher sampling-ratio,

potentially enhancing the quality of the results. Notice that the query workload may not be evenly distributed among all the groups. For the local policy, the main problem becomes distributing a space constraint over all the groups such that the average query performance is maximized. The problem of deciding this size allocation needs future research.

4.7.2 Extension to Other Similarity Functions

Our discussion so far has mainly focused on the edit distance metric. We now generalize the results to the following commonly used similarity measures: Jaccard similarity and cosine similarity. To reduce the size of inverted lists based on those similarity functions, the main procedure of algorithms DiscardLists and CombineLists remains the same. The only difference is that in DiscardLists, to compute the merging threshold T for a query after discarding some lists, we need to subtract the number of hole lists for the query from the formulas proposed in [76]. In addition, for the estimation of the post-processing time, we also need to replace the estimation of the edit distance time with that of Jaccard and cosine time respectively. Figure 4.19 shows the average running time for the DBLP data using variants of the TimeCost$^+$ algorithm for these two functions. The results on the other two data sets were similar and not included here. We see that the average running time continuously decreased when the reduction ratio increased to up to 40%. For example, at a 40% reduction ratio for the cosine function, the running time decreased from 1.7ms to 0.8ms.

The performance started degrading at a 50% reduction ratio and increased rapidly at a ratio higher than 60%. For a 70% reduction ratio, the time for the Cosine and Jaccard functions increased to 150ms and 115ms for LongList. Also, for high reduction ratios the TimeCost and TimeCost$^+$ methods became worse than the panic-based methods, due to the inaccuracy in estimating the merging time. Note that, in our implementation, the Cosine and Jaccard functions are expensive to compute, and therefore the punishment (in terms of



(a) Jaccard function (b) Cosine function

Figure 4.19: Reducing index size on Jaccard and Cosine (DBLP titles)

post-processing time) for inaccurately estimating the merging time can be much more severe than that for the edit distance. The main reason for this performance difference as compared to edit distance is that our edit-distance verification algorithm used early termination, and as a result it was often faster than computing Jaccard or Cosine.

4.7.3 Integrating Several Approaches

So far, we have studied traditional list-compression techniques, our new methods based on discarding lists and combining lists, and the VGRAM technique independently to reduce the index size. Since these methods are actually orthogonal, we could even use a combination of them to further reduce the index size and/or improve query performance. As an example, we integrated CombineLists with Carryover-12.

We first compressed the index using CombineLists approach with a reduction α and then applied Carryover-12 on the resulting index. We varied α from 0 (no reduction for CombineLists) to 60% in 10% increments. The results of the overall reduction ratio and the average query time are shown in the "CL+Carryover-12" curve in Figure 4.20. The leftmost point on the curve corresponds to the case where $\alpha = 0$. For comparison purposes, we also plotted the

results of using the CombineLists alone shown on the other curve. The results clearly show that by using both methods we can achieve higher reduction ratios better query performance than using CombineLists alone. Consider the first point that only uses Carryover-12. It could achieve a 48% reduction with an average query time of 7.3ms. By first using CombineLists at a 30% reduction ratio (4th point on the curve) we could achieve a higher reduction ratio (61%) at a lower average query time (6.34ms).

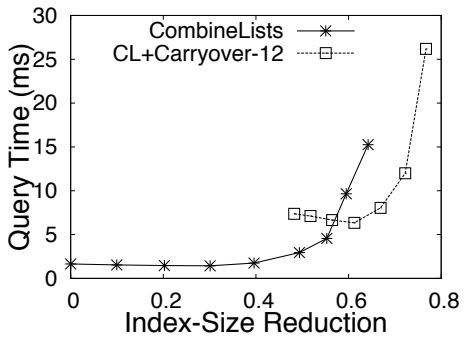


Figure 4.20: Reducing index size using CombineLists with Carryover-12.

One way to integrate multiple methods is to distribute the global memory constraint among several methods. Notice since Carryover-12 and VGRAM do not allow explicit control of the index size, so it is not easy to use them to satisfy an arbitrary space constraint. Several open challenging problems need more future research. First, similar to the space allocation problem discussed in Section 4.7.1, we need to first determine and then distribute the global memory constraint among different methods. Second, we need to decide in which order to use them. For example, if we use CombineLists first, then we will never consider discarding merged lists in DiscardLists, and vice versa.

4.8 Conclusions

In this chapter, we have presented how to reduce the size of a q-gram inverted index to a given space budget for the global alignment version of approximate string queries. We have studied how to adopt existing compression techniques, and we have proposed two novel compression methods: one based on discarding lists, and one based on combining correlated lists. These methods are both orthogonal to existing compression techniques, exploit the unique T-occurrence property of our setting, and offer new opportunities for improving query performance. We studied technical challenges in each method and proposed efficient, cost-based algorithms for solving related problems. Experiments on three real data sets show that our approaches can provide applications with the flexibility to decide the tradeoff between query performance and indexing size and that they can outperform existing compression techniques. An interesting and surprising finding is that while we can reduce the index size significantly (up to 60% reduction) with tolerable performance penalties, for 20-40% reductions we can even improve query performance compared to original indexes.

Chapter 5

External-Memory Methods for Answering Similarity Queries

5.1 Introduction

In this chapter, we study how to answer global-alignment approximate string queries when the data and the indexes reside on disk. Many existing works on approximate string queries have assumed memory-resident data and indexes. The indexes and data in many applications could be so large that even compression cannot shrink them to fit into main memory. On the other hand, even if they fit permanently, dedicating a large portion of memory to them may be unacceptable. For instance, database systems need to deal with many different indexes and concurrent queries, leading to heavy resource contention. In another scenario, consider a desktop-search application supporting similarity queries. It is likely to be unpleasant or infeasible to keep the entire index in memory when desktop search plays a secondary role in the users' daily activities, and hence we must consider secondary-storage solutions.

In recent years, major database systems such as Oracle [7], DB2 [8], and SQL Server [25]

have each added support for some kind of approximate queries. Though their implementation details are undisclosed, they clearly must employ disk-based indexes. Our work could help to improve the performance of similarity queries in these systems.

Contributions: Storing data strings and inverted lists on disk dramatically changes the costs of answering approximate string queries. This setting presents to us new tradeoffs and allows novel optimizations. One solution is to simply map the inverted lists to disk, and use existing in-memory solutions for answering approximate string queries, retrieving inverted lists from disk as necessary. However, as we will see shortly, this simple strategy can be improved upon significantly. We make the following contributions here. In Section 5.3 we propose a new storage layout for the inverted index and show how to efficiently construct it with limited buffer space. The new layout uses the unique filtering property of our global-alignment setting (see Section 5.2) to split the inverted list of a gram into smaller sublists that still remain contiguous on disk. We also discuss controlling the placement of the data strings on disk to improve query performance. Intuitively, we want to order the strings to reflect the access patterns of queries during post-processing. In Section 5.4 we develop a cost-based, adaptive algorithm for answering queries. It effectively balances the I/O costs of accessing inverted lists and candidate answers. It becomes technically interesting how to properly quantify the tradeoff between accessing inverted lists and candidate answers when combining our new partitioned inverted index with the adaptive algorithm. Finally, Section 5.5 presents a series of experiments on large, real datasets to study the merit of our techniques. We show that the index-construction procedure is efficient, and that the adaptive algorithm considerably improves query performance. In addition, we show that our techniques outperform a recent tree-based index, BED-tree [124], by orders of magnitude.

5.1.1 Related Work

There are many studies on disk-based inverted indexes, mostly for information retrieval [126]. Index-construction procedures are presented in [126], and index-maintenance strategies are evaluated in [74]. Such disk-based inverted indexes have been extensively studied for conjunctive keyword queries [126]. For example, a conjunctive keyword query might ask for documents containing the words "cat" and "dog". However, there are several key differences between our problem and the traditional problem of conjunctive keyword queries. First, a keyword query often consists of only a few words, and the opportunity of pruning results is limited to a few inverted lists. In contrast, an approximate string query could have many grams, and it may be inefficient to "blindly" read *all* their inverted lists. Second, for conjunctive keyword queries an answer should typically appear on all the inverted lists of the query's keywords. In contrast, for an approximate string query, an answer does not necessarily occur on all the query grams' inverted lists. Third, keyword queries ask for possibly large documents, whereas approximate string queries ask for strings relatively smaller than documents. This difference in the size of answers makes it more attractive in our setting to ignore some inverted lists and pay a higher cost in the final verification step.

Compression can improve transfer speeds of disk-resident structures. There are various compression algorithms tailored to inverted lists [126, 12, 127, 73, 22]. These methods are related but orthogonal to the techniques described in this chapter.

5.2 Preliminaries

5.2.1 Pruning with Filters

Pruning candidate answers based on the T lower bound as discussed previously (Section 2.4.1) is referred to as the "count filter" [44]. We briefly review a few of the many other important filters which are mostly specific to the global alignment version of approximate string matching.

Length Filter: An answer to a query with string s and edit distance k must have a length in $[|s| - k, |s| + k]$ [44]. Analogous findings exist for other similarity functions [76]. For example, we can immediately see that the strings "`cat`" and "`cathey`" cannot be within edit distance 1 simply because their length difference is 2.

Prefix Filter: Suppose we impose a global ordering on the universe of grams. Consider a query with an ordered gram set $G(r)$ and lower bound T. An answer must share at least one gram with the query in its "prefix". That is, the first $|G(r)| - T + 1$ grams of an answer r must share at least one gram with the first $|G(r)| - T + 1$ grams of the query [28]. For example, consider a query "`cathey`" with 2-grams $\{\texttt{ca}, \texttt{at}, \texttt{th}, \texttt{he}, \texttt{ey}\}$ and an edit-distance threshold of 1, yielding a lower bound $T = 3$. Further, suppose we used a lexicographical global ordering of all grams, then the ordered gram set of "`cathey`" is $\{\texttt{at}, \texttt{ca}, \texttt{ey}, \texttt{he}, \texttt{th}\}$. According to the prefix filter any string within edit distance 1 of "`cathey`" must have at least one of the grams in "`cathey`"'s prefix gram set $\{\texttt{at}, \texttt{ca}, \texttt{ey}\}$.

Other Filters: Some filters exploit the the positions of g-grams, e.g., the position filter [44] and content-based filter [120], or mismatching q-grams [120] for pruning. For example, consider the strings "`cathey`" and "`theyca`" with 2-gram sets $\{\texttt{ca}, \texttt{at}, \texttt{th}, \texttt{he}, \texttt{ey}\}$ and $\{\texttt{th}, \texttt{he}, \texttt{ey}, \texttt{yc}, \texttt{ca}\}$, respectively. The strings' 2-gram sets share 4 grams which means that according to the $T = 3$ lower bound the strings could be within an edit distance of 1. How-

ever, based on the starting positions of the 2-grams in their strings, we can clearly see that the strings cannot be within edit distance 1. Intuituvely, two grams "match" only if their positions differ by no more than the edit-distance threshold.

5.2.2 Filter Tree

We can use filters to partition data strings into groups. We use a structure called FilterTree [76] to facilitate such a partitioning. Originally, the FilterTree was designed for in-memory inverted lists. In Section 5.3.2 we discuss how to place the inverted lists onto disk.

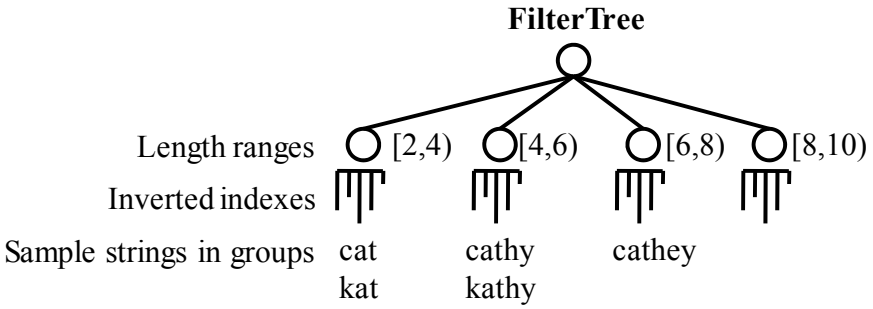


Figure 5.1: A FilterTree partitioning data strings by length.

Figure 5.1 shows a FilterTree that partitions the data strings by their length. A leaf node corresponds to a range of lengths (called group), and each string belongs to exactly one group. For each group we build an inverted index on the strings in that group. To answer a query we only need to process some of the groups. For example, the answers to a query with string "`cathey`" and edit distance 1 must be within the length range [5, 7]. To answer the query, we only access the inverted indexes of the *relevant* groups [4,6) and [6,8).

5.3 Disk-Based Index

In this section we introduce a disk-based index to answer global-alignment approximate string queries. We detail its individual components, and discuss how to construct and store them on disk.

5.3.1 System Context

Suppose we have a table about persons with attributes such as address, name, phone number, as shown in Figure 5.2. We want to create an index on the "Name" attribute to support similarity selections. All components except the source table belong to this index and are intended to be part of the database engine. The components below the dotted line are stored on disk, and only the FilterTree is assumed to fit in memory.

Dense Index: The lowest level of our index is a *dense index* that maps names to their record ids in the source table. Each entry in the dense index is identified by a unique id, called string id (SID). Our decision to project the indexed column into a dense index is motivated by the following observations. (1) The number of candidate answers could be much higher than the number of true answers. Since the tuples in the source table could have many attributes, it would be inefficient to retrieve them for removing false positives. Instead, we use the dense index for post-processing and only access the source table for true answers. (2) The additional level of indirection allows us to choose the physical organization of the dense index independently of the source table, which can improve query performance (see Section 5.3.4).

Inverted Index and FilterTree: The upper level of our index consists of an in-memory FilterTree (Section 5.2.2) and a corresponding disk-resident inverted index. Each leaf in the FilterTree has a map from each of its grams to an inverted list address (l, o), indicating the

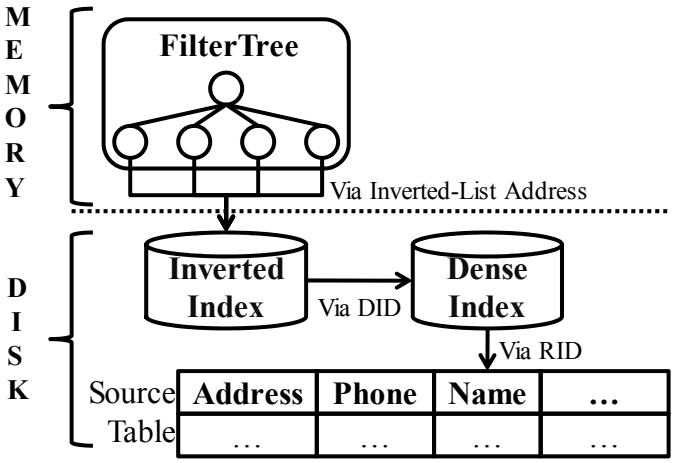


Figure 5.2: Components of an index on the "Name" field of a "Person" table.

offset o and length l of a gram's inverted list in a particular group. The inverted index stores inverted lists of string ids (SIDs) in a file on disk. Typically, the FilterTree is very small (a few megabytes) compared to the inverted index (hundreds of megabytes or gigabytes).

Answering Queries: To answer a query, we first use the FilterTree to identify all *relevant* groups according to the partitioning filter, e.g., groups that are within a certain length range. Then, for each relevant group we retrieve the inverted lists corresponding to the query tokens, and solve the T-occurrence problem (Section 5.2) to identify candidate answers. To remove false positives, we retrieve the candidates from the dense index and compute their real similarity to the query. In further discussions we ignore the final step of retrieving the records from the source table since this step cannot be avoided, and we want to keep our solutions independent of the organization of the source table.

5.3.2 Partitioning Disk-Based Inverted Lists

Inverted lists are often stored contiguously on disk [126] so they can be read sequentially. This layout maximizes selection-query performance at the expense of more costly updates. Let us examine Figure 5.3 to discuss possible layouts of inverted lists partitioned using a

filter.

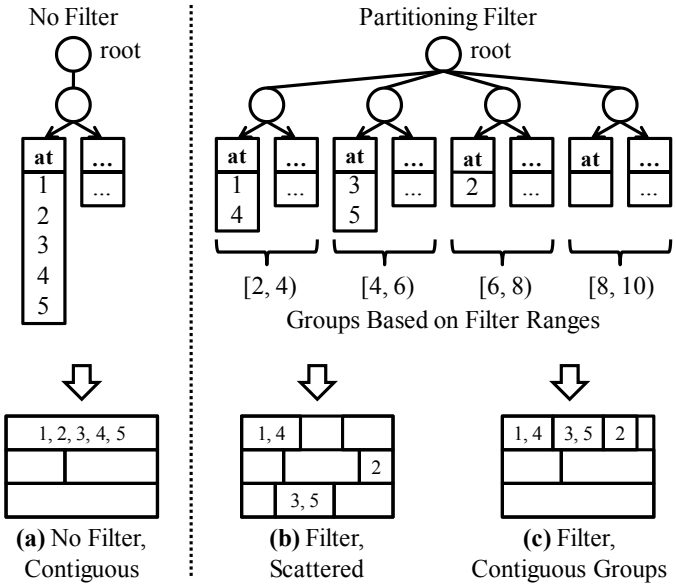


Figure 5.3: Index partitioning and physical organizations. The inverted list of gram "at" with and without partitioning and possible mappings to a file on disk. Organization (c) is the best.

In the left upper portion we see a gram at's inverted list without partitioning. To the right, the original inverted list is split into different groups according to a filter. At the bottom we show possible storage layouts. Without partitioning we can simply store the list contiguously as shown in organization (a). In organization (b), each inverted list is stored contiguously, but the lists belonging to the same gram from different groups are scattered across the file. Finally, organization (c) places the inverted lists belonging to the same gram contiguously in the file.

Example: Suppose we partitioned the strings by length, as shown in Figure 5.3. Consider a query with the string "cathey" and an edit-distance threshold $k = 1$. Let us examine the cost for retrieving the inverted list of gram at. In organization (a), we perform one disk seek and transfer the five elements $\{1, 2, 3, 4, 5\}$ for at's inverted list. With filtering, answers to the query must be within the length range $[5, 7]$, since the length of the query string "cathey" is six. Therefore, only the groups $[4, 6)$ and $[6, 8)$ are relevant. Using organization

(b) we transfer three list elements $\{2, 3, 5\}$ with two disk seeks. Note that the elements $\{3, 5\}$ and $\{2\}$ could be arbitrarily far apart in the file. Using organization (c) we transfer the same three elements $\{2, 3, 5\}$ but with only one disk seek.

5.3.3 Constructing Inverted Index on Disk

We now present how to efficiently build an inverted index in the physical layout shown in Figure 5.3(c). Since we want to index large string collections, we cannot assume that the final index fits into main memory. Hence, we desire a technique that (1) can cope with limited buffer space, and (2) scales nicely with increasing buffer space. Our main idea is to first construct an unpartitioned inverted index with a standard merge-based method [126] (Phase I), then reorganize the inverted lists to reflect partitioning (Phase II). This two-phase approach has the advantage that any existing construction procedure for an inverted index can be directly used in Phase I without modification. Later in this subsection, we discuss an alternative construction procedure that directly modifies a standard merge-based technique. Figure 5.4 shows both phases of index construction, described as follows.

Phase I: This is a standard merge-based construction procedure [126]. We read the strings in the collection one-by-one, decompose them into grams, and add the corresponding string ids into an in-memory inverted index. Whenever the in-memory index exceeds the buffer limitation, we flush all its inverted lists with their grams to disk, creating a run. Once all strings in the collection have been processed, we combine all runs into the final index in a merge-sort-like fashion.

Phase II: This phase reorganizes the inverted index constructed in Phase I into a partitioned inverted index better suited for approximate string queries. We start with an "empty" FilterTree, whose leaves contain empty mappings from grams to list addresses. We assign a unique id to each leaf (LID). Then, we sequentially read lists from the merged index until

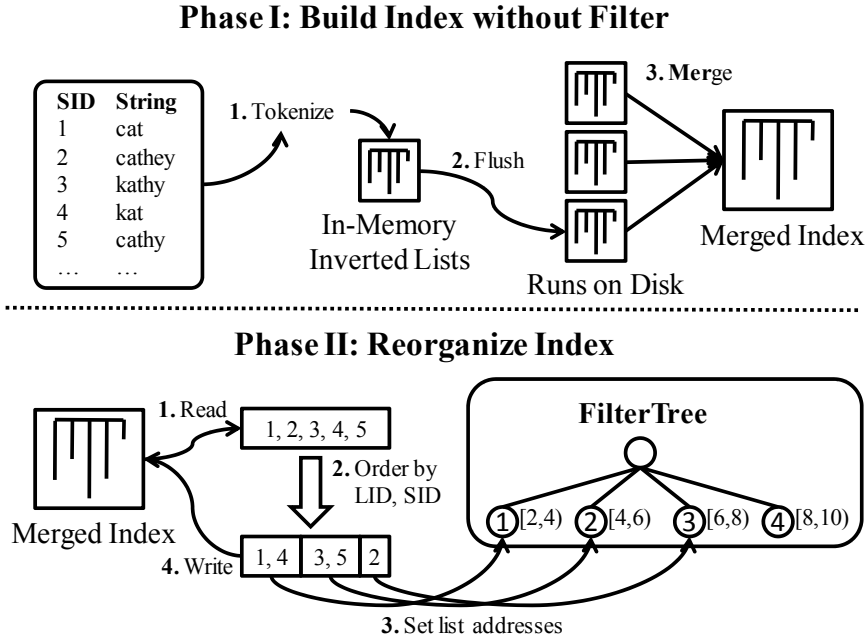


Figure 5.4: Two phases of index construction. Phase I builds the index without partitioning. Phase II re-organizes the index to reflect partitioning.

the buffer is full. For each of those lists, we re-order its elements based on their FilterTree leaf ids (LID) and string ids (SID). The re-ordering can be supported by a map from SID to LID created in Phase I, or we can compute the corresponding $LIDs$ on-the-fly to save memory. In the process, we set the lengths and offsets (list addresses) for the lists of different groups in the FilterTree. After the elements of each list in the buffer have been re-ordered, we write these lists in the buffer back to disk, in-place, in one sequential write. We repeat these steps until all the lists of the merged index have been processed. Notice that the cost of Phase II is independent of the number of groups used in partitioning. If the merged index has m bytes and the buffer holds b bytes, then Phase II requires $2 * \frac{m}{b}$ sequential I/Os plus the CPU cost for sorting. The total cost is governed only by the index size and the buffer size, thus Phase II meets the performance requirements.

Integrating Partitioning in Phase I: Instead of reorganizing an unpartitioned index in a separate phase (Phase II), we could directly modify the merge-based construction procedure to build a partitioned index as follows. When merging the runs on disk we immediately

reorder the elements of a completely assembled inverted list before writing it to the final index, similar to the reordering done in Phase II. We also need to update the mappings in the FilterTree accordingly. The advantage of this construction procedure is that it eliminates the reorganization phase (Phase II). However, the procedure will need to be either written from scratch, or integrated into an existing merge-based construction algorithm.

Dealing with Large Edit Distances: To answer queries when the lower bound $T \leq 0$ we cannot rely on the grams for pruning. We would usually resort to scanning the dense index and computing the real similarity of each entry to the query. To improve the performance of such queries, for each group generated by the partitioning filter, we maintain an extra list of all the distinct string ids in that group. We store these lists on disk using organization (c), meaning that the lists of different groups will be adjacent in the file. For queries with $T \leq 0$ we read the list of distinct string ids for all relevant groups in one sequential I/O. Then we retrieve only those strings from the dense index and compute their similarities to the query (as opposed to retrieving *all* strings). This technique is complemented by a good physical layout of the dense index.

5.3.4 Placing Dense Index on Disk

Next we study different ways to organize the entries in the dense index. Recall that to answer a query we use the inverted lists to get candidate answers. Then, for each candidate id, we access the dense index to retrieve its corresponding $\langle String, RID \rangle$ pair and compute its similarity to the query (Section 5.3.1). Intuitively, we want those entries that are likely to be accessed by the same query to be physically close on disk. That is, we want to minimize the number of blocks that need to be accessed to retrieve a given set of candidates. At the same time we want to avoid seeking through the entire dense-index file to find the candidates. Note that these desiderata are different, e.g., even if we only needed to retrieve a few blocks,

those requested blocks could be far from each other in the file. The following are two natural ways to organize the entries:

Filter Ordering: Partitioning filters offer a simple but effective solution. If we sort the entries of the dense index by a filter, then we limit the range of blocks in which candidates of a query could appear. Also, it is more likely for blocks to contain entries that are similar to each other, and hence, similar to a query. For example, we can limit the blocks that could contain answers to a query "`cathey`" with edit distance 1 to those that contain strings of length [5,7]. Within that range we may perform random accesses to get the needed blocks, but the seek distances between these blocks are relatively small.

Clustering: Another solution is to group similar entries together in the file by running a clustering algorithm on them (e.g., k-Medoids [64]). This global solution has drawbacks. (1) Clusters accessed by the same query could be arbitrarily far apart in the file. (2) The clustering process could be very expensive. (3) Updates to the index would be complicated. It is also possible to combine a filter ordering with clustering (hybrid), e.g., we could sort the strings by their length, and then we run a clustering algorithm within each length group.

We implemented all the above organizations, i.e., filter orderings, clustering using k-Medoids, and a hybrid approach. We experimentally found that all of the approaches perform much better than an arbitrary ordering of the strings. The more sophisticated methods, clustering and hybrid clustering, did not offer better query performance than a simple filter ordering. Therefore, we prefer the filter ordering solution.

5.3.5 Index Maintenance for Updates

Inverted Index: Standard techniques [74] for inverted-index maintenance can be used to update our partitioned inverted index. They mostly involve buffering insertions into an in-

memory index, and deletions into an in-memory list. Periodically the in-memory structures are integrated into the index on disk, e.g., using one of the following strategies [74]:

In-Place: For each list in the in-memory index, read the corresponding list from disk, combine the two lists, and write the new list back to disk (possibly to a different location).

Re-Merge: Read the lists in the file one-by-one. Whenever a list has pending changes in the in-memory index, update the list. Finally, write all (possibly updated) lists into a new file.

We can apply such maintenance strategies to our new physical layout from Section 5.3.2. We add a FilterTree structure to the in-memory index that buffers updates. Alternatively, we could omit the FilterTree, and deal with reorganizing the inverted lists to reflect partitioning when we combine the in-memory index with the one on disk.

Dense Index: The dense index is a sorted sequential file that allows various implementations and associated maintenance strategies. For example, we could simply use a B+-tree. Since updates to the inverted index are done in a "bulk" fashion, it would be prudent to update the dense index (B+-tree) in a similar way to maintain the sequentiality of (leaf-) entries. For example, bulk-loading a new B+-tree or using an extent-based B+-tree would achieve this goal.

5.4 Cost-Based, Adaptive Algorithm To Answer Queries

In this section we present a cost-based, adaptive algorithm for answering similarity queries using the disk-based index. For simplicity, we first describe the algorithm assuming the inverted-index organization in Figure 5.3(a), and later discuss how to modify it to work with organizations (b) and (c).

5.4.1 Intuition

Recall that to answer a query we tokenize its string into grams, retrieve their inverted lists, solve the T-occurrence problem to get candidate answers, and finally retrieve those candidates from the dense index to remove false positives. We develop the algorithm based on the following observations. (1) Candidate answers must occur on at least a certain number of the query grams' inverted lists, but not necessarily on all those inverted lists. (2) The pruning power of inverted lists comes from the *absence* of string ids on them. For example, an inverted list that contains all string ids cannot help us remove candidates. Intuitively, long lists are expensive to retrieve from disk and may not offer much pruning power. An approximate query with a string r, q-grams $G(r)$, and an edit-distance threshold k must share at least $T = |G(r)| - k * q$ grams with an answer, with $|G(r)| = |s| - q + 1$. According to the prefix filter (Section 5.2.1), the minimum number of lists we must read to ensure the completeness of answers is $minLists = |G(r)| - (T - 1)$. To understand this equation, consider a string id that occurs on $T - 1$ lists. To become a candidate, it must additionally occur on at least one of the other $|G(r)| - (T - 1)$ lists. For example, if $q = 2$ and $k = 1$, a query with string "`cathey`" has $|G(r)| = 6 - 2 + 1 = 5$ grams, $T = 5 - 1 * 2 = 3$ and $minLists = 5 - (3 - 1) = 3$.

At one extreme, we could read just the $minLists$ shortest lists. As a consequence, the number of candidates for post-processing could be high, because every string id on those lists needs to be considered. Recall that for post-processing we retrieve strings from the dense index (Figure 5.2), hence every candidate could require a random disk I/O. At the other extreme, we could read all the inverted lists for a query, including those long and expensive ones with little pruning power. It is natural to strive for a solution between the two extremes, balancing the I/O costs for retrieving inverted lists and for probing the dense index.

5.4.2 Algorithm Details

To answer a query we begin by reading the $minList$ shortest inverted lists (corresponding to grams in the query) from disk into memory. Then we traverse these lists to obtain a set of initial candidates containing all the lists' string ids (using an algorithm such as "HeapMerge" [104]). The set of initial candidates, denoted by $C = \{c_1, c_2, \ldots, c_{|C|}\}$, is a set of triples $c = (sid, cnt, cnt_a)$. In each triple, sid denotes the string id of the candidate, cnt denotes the current "count" or the total number of occurrences of sid on the lists we have read so far, and cnt_a denotes the "count absent" value (its meaning will become clear shortly). Next, we decide whether we can reduce the overall cost of the query by reading additional lists to prune candidates. If so, then we read the next list and prune candidates with it. If not, then we post-process the candidates in C. The decision whether or not to read the next list depends on the current number of candidates still in C, the cost for post-processing them by accessing the dense index, the cost for reading more lists, and the likelihood of pruning candidates by reading more lists. We repeat this process until we have read all the query grams' inverted lists or there is no cost reduction from reading additional inverted lists.

Note that the benefits of reading additional inverted lists might manifest themselves after reading a few more lists, and not necessarily after reading only one more list. Hence, we must estimate the effects of reading the next λ lists. Take our running example, with $r =$ `cathey`, $k = 1$, and lower bound $T = 5$ as shown in Figure 5.5. Let us examine candidate $c_1 = (2, 2, 2)$, which is part of the initial candidates produced after processing the $minLists = 3$ shortest lists. We have read 3 of the 5 lists, so there are 2 lists left. In order to prune candidate c_1, its string id must be absent on at least $cnt_a = 2$ additional lists.

In general, the "count absent" value is the minimum number of additional lists a candidate string id must be *absent* from, in order to be pruned. At each stage in the adaptive algorithm

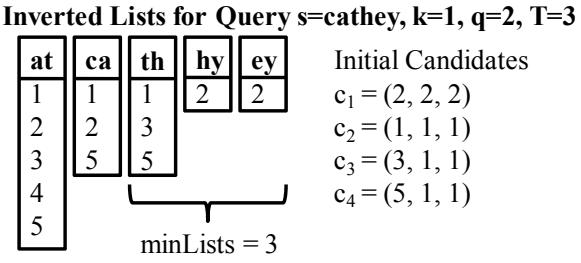


Figure 5.5: Example to illustrate meaning and effect of the "count absent" value.

for answering a query with string r and q-grams $G(r)$, we can compute the count-absent value of a triple (sid, cnt, cnt_a) corresponding to a candidate answer by:

$$cnt_a = (|G(r)| - minLists) - (T - cnt) + 1. \tag{5.1}$$

Candidates could have various cnt_a values, and the cost and benefit of reading more lists depend on λ, i.e., the number of additional lists we consider reading. Since the benefits of reading more lists can become apparent only after reading several next lists, a cost model that considers only reading the next list could get stuck in a local minimum, and is therefore insufficient to find a globally minimal cost. This is the reason why we need to consider reading all the possible numbers of remaining lists λ. For example, in the first iteration we would consider $1 \leq \lambda \leq |G(r)| - minLists$.

Pseudo-Code: Figure 5.6 shows pseudo-code of the adaptive algorithm. We use "$\Omega(r)$" to denote the list addresses sorted by length in an increasing order for a query string r. For convenience we use "$\Omega(r)[j]$" to mean the j-th list in $\Omega(r)$. C_λ refers to the subset of candidates with $cnt_a \leq \lambda$. We assume the following parameters of a cost-model: (1) Θ is the average cost of post-processing a candidate, (2) $\Pi(l)$ is the cost of reading list l, and (3) $ben(\lambda)$ is the benefit or reading λ additional lists.

We begin by reading $minLists$ inverted lists and processing them to obtain a set of initial candidates (line 1). Next, we consider reading all numbers of remaining lists, setting λ

```
Input: Inverted list addresses Ω(r) for a query string r
       Minimum number of lists to read minLists
Output: A set C of candidate answers to be post-processed
Method:
1.  C = HeapMerge(Ω(r), minLists);
2.  nextList = minLists + 1;
3.  listsLeft = |Ω(r)| − minLists;
4.  WHILE listsLeft > 0
5.      FOR λ = 1 TO listsLeft
6.          Λ = next λ lists starting from Ω(r)[nextList];
7.          costPP = |C_λ| ∗ Θ; // cost of post-processing
8.          costRL = Σ_{l∈Λ} Π(l); // cost of reading lists
9.          benfitRL = ben(λ) for lists in Λ;
10.         IF (costRL - benefitRL < costPP) THEN
11.             invList = readList(Ω(r)[nextList]);
12.             C = pruneCandidates(C, invList);
13.             nextList+=1; listsLeft−=1;
14.             BREAK FOR;
15.         END IF
16.     END FOR
17.     IF λ == listsLeft THEN BREAK WHILE; // no benefit
18. END WHILE
19. RETURN C;
```

Figure 5.6: Cost-based, adaptive algorithm for answering queries.

accordingly. We compute the cost of post-processing the candidates that could potentially be pruned with λ lists (line 7). Correspondingly, we compute the cost (line 8) and benefit (line 9) of reading the next λ lists. We decide whether we can reduce the overall cost by reading λ lists, as compared to not reading them (line 10). If we can reduce the cost, we read the next list (line 11) and prune candidates (line 12). Otherwise, we proceed with the next λ. We repeat this process until euther we have read all lists, or we cannot benefit from reading more lists. Notice that whenever we detect a benefit we read just *one* more list at a time, independent of the value of λ. This approach mitigates possible inaccuracies of a real cost-model.

5.4.3 Cost Model

In this section we develop a model to estimate the cost and benefit of reading additional inverted list.

Cost of post-processing candidates: We focus on the I/O cost because the CPU time for computing similarities is often negligible in comparison. The maximum cost for retrieving a candidate string is the time for one disk seek plus the time to transfer one disk block. This cost can be determined directly from the hardware configuration or estimated offline. We can improve this model by taking into account the chance that a dense index block is already cached in memory. Using a conservative but simple approach, we read a few blocks offline and compute the average time to retrieve a block, denoted by Θ. We then use $|C| * \Theta$ to estimate the cost of post-processing a candidate set C. For example, the cost of post-processing the candidates in Figure 5.5 would be 4*Θ since $|C| = 4$.

Cost of retrieving inverted lists: Our experiments showed that the lengths of inverted lists usually follow a Zipf distribution, thus the average retrieval time is not an accurate estimator of the true cost. Again, the cost is determined by the seek time and the transfer rate of the disk. Since our solution is on top of a file system, the raw disk parameters are not very accurate performance indicators either, due to the intermediate layer. To overcome this issue, we read a few lists offline and do a linear regression to obtain the cost function $\Pi(\omega) = m * \omega.l + b$, where $\omega.l$ denotes the length of the inverted list ω, b is an estimate for the seek time, and m is an estimate for the inverse transfer rate. So, given a set Ω of ω, we can estimate the total cost of reading all those inverted lists by $\sum_{\omega \in \Omega} \Pi(\omega)$. For example, the cost of reading the lists of grams `ca` and `at` from Figure 5.5 would be $(m * 3 + b) + (m * 5 + b)$, since their lengths are 3 and 5, respectively.

Benefit of reading additional inverted lists: At a high level, we quantify the benefit in terms of the cost we can save by pruning candidates, considering the likelihood of that

happening by reading λ lists. The likelihood of pruning a candidate c with λ lists depends on its cnt_a. Intuitively, we would like to know the probability that c's string id is absent from at least cnt_a of the λ lists. Computing this probability for each candidate individually would be computationally expensive. To avoid repeated computations we group the candidates by cnt_a as follows. We define a subset of C as $C(i) = \{c|c \in C \wedge c.cnt_a = i\}$, containing all candidates in C that have a certain $cnt_a = i$. We also make the following simplifying assumptions:

- The probability of one string id being present or absent from an inverted list is independent of the presence or absence of another string id on the same list.
- The probability of a string id being present or absent from one inverted list is independent of that same string id being absent or present from another list.

Following these assumptions, we estimate the benefit as:

$$ben(\lambda) = \Theta * \sum_{i=1}^{\lambda} |C(i)| * p(i, \lambda), \tag{5.2}$$

where $p(i, \lambda)$ denotes the probability of a string id being absent from i of the λ lists. In a sense, we are being optimistic, since Equation 5.2 expresses that we could prune *all* candidates in $C(i)$ with probability $p(i, \lambda)$. The key challenge is to obtain a reasonably accurate $p(i, \lambda)$ efficiently. Following our assumptions we model $p(i, \lambda)$ as a sequence of λ independent Bernoulli trials, i.e., a Bernoulli process [93]. We define "success" as the absence of a string id on a list, and "failure" as the presence of a string id on a list. Since all we know of the lists are their lengths, we estimate the success probability with a set Ω of list addresses ω as:

$$p_s = 1 - \frac{\sum_{\omega \in \Omega} \omega.l}{|\Omega|} * \frac{1}{N}. \tag{5.3}$$

The term $\frac{\sum_{\omega \in \Omega} \omega.l}{|\Omega|}$ denotes the average list length of the λ lists in Ω we are considering to read, and N is the total number of strings in the dense index. The probability of at least

i successes in λ trials having a success probability p_s can be computed with the cumulative binomial distribution function:

$$p(i, \lambda) = binom(i, \lambda, p_s) = \sum_{j=i}^{\lambda} \binom{\lambda}{j} p_s^j (1 - p_s)^{\lambda - j}. \tag{5.4}$$

Combining Equations 5.2, 5.3 and 5.4 yields a complete model for estimating the benefit of reading λ additional lists:

$$ben(\lambda) = \Theta * \sum_{i=1}^{\lambda} |C(i)| * binom(i, \lambda, p_s). \tag{5.5}$$

5.4.4 Combining with Partitioned Inverted-Index Layout

So far, we have presented the adaptive algorithm using the layout in Figure 5.3(a) from Section 5.3 for simplicity. Next, we discuss how to combine the adaptive algorithm with the other two inverted-index layouts (b) and (c). Recall that when answering a query we first traverse the FilterTree to identify the relevant groups (e.g., corresponding to length ranges) that could contain answers.

Modified Success Probability: In Equation 5.3 the probability (called success probability) of a string id being absent from a set of lists depends on their average list length, and the total number of strings N in the dataset. Since each group generated by a partitioning filter only contains a subset of the N total strings, simply applying Equation 5.3 would lead to overestimation of the success probability, and consequently overestimation of the benefit (Equation 5.5). We modify the success probability by replacing N in Equation 5.3 with the number of strings in the particular group we are considering. This change applies to both inverted-index layouts (b) and (c). Next, we detail how to answer queries with those two layouts.

Layout (b): To answer a query using the adaptive algorithm, we first identify all relevant groups, and then process each group individually as if there was no partitioning (but with the new success probability). Each instance of the algorithm tries to minimize the cost for processing its corresponding group.

Layout (c): Intuitively, we just process all relevant groups together, as opposed to processing them one-by-one. We first read the *minList* inverted lists of all relevant groups to create the sets of initial candidates for *all* groups at the same time. To exploit the contiguous layout, we retrieve the lists gram by gram. That is, we read the lists of all relevant groups of the first gram, then we read the lists of all relevant groups of the second gram, etc., until we have retrieved all the *minLists* lists. We perform HeapMerge on the groups separately to get the initial sets of candidates.

Next, we proceed with the iterative phase of the algorithm, estimating the cost and benefit of reading the next grams' lists of all relevant groups. We either decide to read the next lists together or commence post-processing.

Layout (c) Examples: In the following, we describe how to handle a few interesting scenarios when combining the adaptive algorithm with organization (c). For example, Figure 5.7 shows the inverted lists of the relevant groups for our running example `cathey` (the string ids differ from earlier examples). To minimize the global cost we should read the lists for gram `hy` first because there are a total of 3 relevant elements {2, 6, 8}. Next, we should read the lists of gram `ey` because they contain a total of 4 elements {2, 6, 7, 8}, and so on.

Local versus Global Ordering: We want to read the inverted lists from shortest to longest. However, the local order of each group may not correspond to the best global order. We must read the relevant lists of grams in order of their sum of lengths.

Pruning Entire Groups: Groups that do not contain at least T grams of the query can be pruned entirely. We should not read inverted lists of such pruned groups.

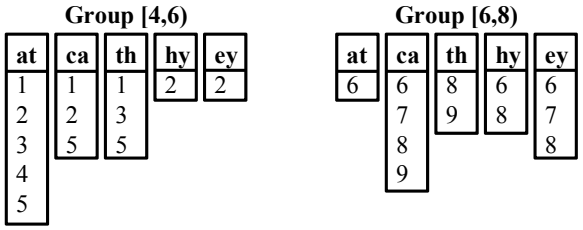


Figure 5.7: Inverted lists for query `cathey`, $k = 1$, $q = 2$, and $T = 3$.

Figure 5.8 shows a group [6,8) that has less than T of the query's grams. Since this group cannot contain answers to the query, we do not need to read lists from it. For example, for gram `ca` that exists in group [6,8), we should only read the elements {1,2,5} and *not* the elements {6,7,8,9}.

A similar situation can arise during the iterative phase of the adaptive algorithm. When a particular group does not contribute candidates anymore (because they have all been pruned), we do not need to read inverted lists from that group anymore (but possibly still from the other ones). We must handle both scenarios above specially during cost estimation and during the retrieval of lists.

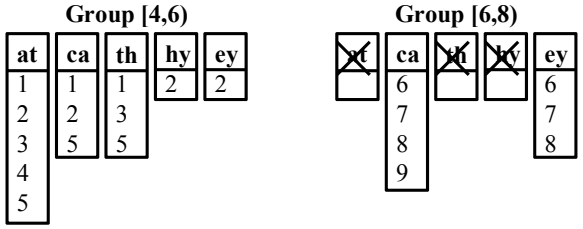


Figure 5.8: Example of pruning entire groups. Inverted lists for query `cathey`, $k = 1$, $q = 2$, and $T = 3$. None of the strings in group [6,8) contain the grams `at`, `th`, or `ty`.

5.4.5 Removing Candidates Early

The number of initial candidates generated from the *minList* shortest lists could be high. Post-processing these candidates or reading more inverted lists to prune them could incur

additional disk accesses. Next, we briefly discuss a few additional filtering techniques to reduce the number of candidates, especially after reading the *minList* lists. Having fewer candidates means (1) we need to access fewer dense-index blocks for post-processing, and (2) the adaptive algorithm might choose to read fewer inverted lists.

In-Memory Signatures: One way is to keep summary information of each candidate in memory for pruning purposes. The rationale is that it may be worth spending the extra memory to save disk accesses. The most promising approach we implemented is based on unigrams (q-grams with $q = 1$), called "letter count" filter [62]. The main idea of this filter is to derive a lower edit-distance bound on two strings by examining the differences of their character (unigram) frequencies. Originally, the filter assumes all characters of both strings are known. In our scenario, we clearly cannot keep *all* characters of the data strings in memory. Intuitively, we only wish to keep the frequencies of the d most discriminating characters for each string, where d is a small constant depending on how much memory we are willing to spend. We estimate which characters are most discriminating as follows. We first compute the total frequency of each character in the dataset. Since the most frequent characters are not neccessarily the most discriminating ones (because most strings could contain a similar amount of them), we additionally consider the standard deviation of character frequencies in the dataset. That is, the most discriminating characters are those that have the highest product of total frequency and standard deviation. In Section 5.5 we experimentally show the benefit of this letter count filter. In most cases keeping only the 4 or 5 most discriminating characters is a good choice.

Positional Information: Another way to achieve higher pruning power is to enhance the inverted lists with additionl information, e.g., the positions of q-grams. Positional information enables the use of other filters, e.g., the positional filter [44] and the mismatch and content-based filters [120]. Another idea is to use the positions to partially reconstruct a candidate's string, and run the classic dynamic programming algorithm to get a lower bound

on the edit distance between the query and that candidate, but with "wildcards" for those unknown characters of the candidate. As we read more inverted lists, we add more characters to the partial string of each candidate, and possibly update the edit-distance lower bound. We implemented all of the positional approaches above, and our experiments showed that the additional pruning power did not outweigh the higher cost for retrieving the enhanced inverted lists. We omit the details of these experiments for brevity, and leave this topic for future research.

5.5 Experiments

In this section, we evaluate the performance of queries and index construction of our techniques. We show experimental results on range and top-k queries using edit distance, and range queries using normalized edit distance, Jaccard based on q-grams, and Jaccard based on word grams. We compare our techniques to a recent tree-based index, BED-tree [124], whenever possible.

Datasets: We used six real datasets, summarized in Table 6.5. The first four are taken from [124] (BED-tree) to establish a fair comparison. In addition, we used the last two datasets for experiments on scalability and index-construction since they are larger than the first four.

Dataset	**# Strings**	**Maximal Len**	**Average Len**
DBLP Author	2,948,929	48	15
DBLP Title	1,158,648	667	68
IMDB Actor	1,213,391	73	16
Uniprot	508,038	1992	341
Medline Titles	10,000,000	252	84
Web Word Grams	20,000,000	163	23

Table 5.1: Datasets and their statistics.

The DBLP Title and Author datasets were taken from DBLP[1], and contained authors and titles of publications. The IMDB Actor[2] dataset consists of actor names from movies. The Uniprot[3] dataset contains protein sequences in text format. The Medline Titles[4] dataset consists of publication titles from the medical domain. Finally, the Web Word Grams[5] dataset contains popular word-grams from Web documents. We randomly picked 10 million titles of the Medline Dataset, and 20 million 4-word-grams for the Web Word Grams dataset.

Hardware and Compiler: We conducted all experiments on a machine with a four-core Intel Xeon E5520 2.26Ghz processor, 12GB of RAM, and a 10,000 RPM disk drive, running the Ubuntu 8.04 operating system. We used the original code for BED-tree, written in C++, that the authors generously provided to us. We implemented all our algorithms in C++ as well. We compiled all code with GCC using the "`-O3`" flag.

Parameters: We experimented with different q for tokenizing strings into q-grams and found $q = 3$ to be best for most cases. Therefore, we used $q = 3$ for all experiments, for both BED-tree (where applicable) and for our techniques. For our techniques, we used the length filter for partitioning. We used a disk-block size of 8KB for both BED-tree and our methods.

Clearing Filesystem Cache: In the experiments we considered both the raw disk performance of queries, and their performance with caching. To simplify the implementations, both BED-tree and our techniques were built on top of a filesystem (as opposed to using the raw disk device). Using a filesystem, however, complicates accurate measurements of disk performance due to filesystem caching (it will aggressively use all available memory to cache disk pages). To overcome this issue, we cleared the filsystem cache at certain points (to be explained) with the following shell command:

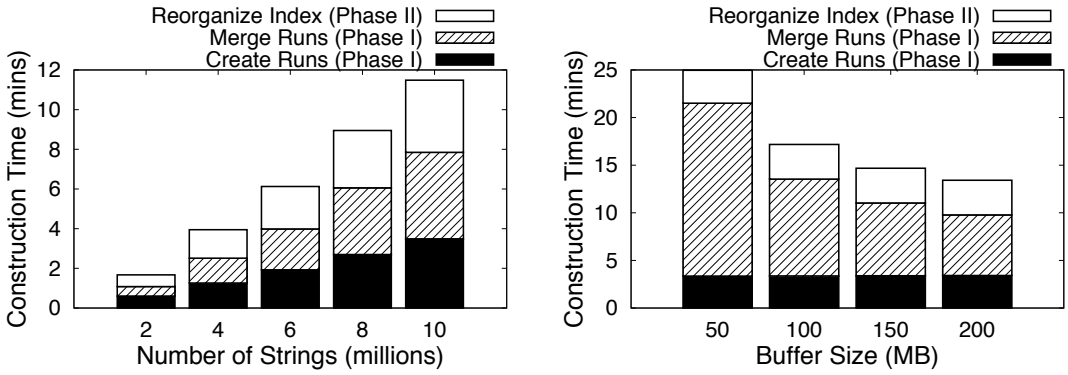
[1] www.informatik.uni-trier.de/~ley/db
[2] www.imdb.com
[3] www.uniprot.org
[4] www.ncbi.nlm.nih.gov/pubmed
[5] www.ldc.upenn.edu/Catalog, number LDC2006T13

```
echo 3 > /proc/sys/vm/drop_caches
```

Query Workloads: The authors of the BED-tree provided us with the data and workloads from their experiments in [124]. The workload for each dataset consisted of 100 randomly chosen strings. For the other datasets used only in this chapter (Medline Titles, Web Word Grams), we also generated workloads by randomly choosing 100 strings for each dataset.

5.5.1 Index-Construction Performance

We built inverted indexes for the Medline and Web Word Grams datasets in organization (c) (Section 5.3) using length filtering. We measured each step of the construction procedure: (1) creating runs, (2) merging the runs, and (3) reorganizing the index, clearing the filesystem cache before each step.

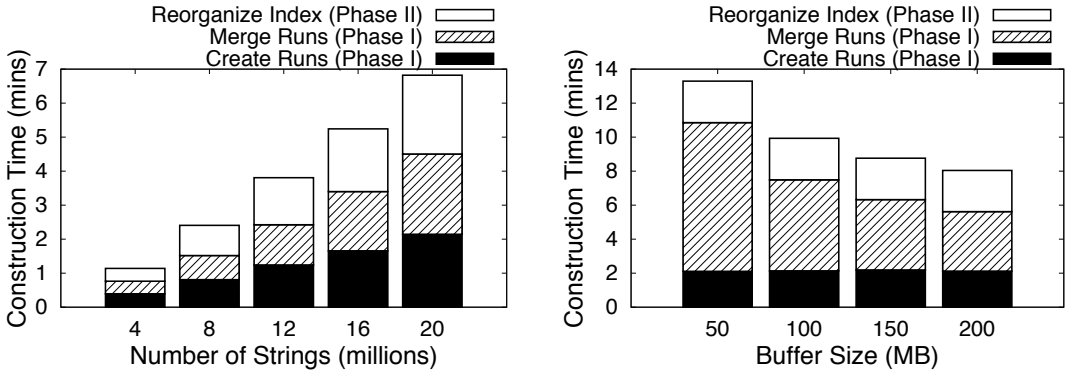


(a) 400MB Buffer.

(b) 10 million strings.

Figure 5.9: Index construction performance on Medline Titles.

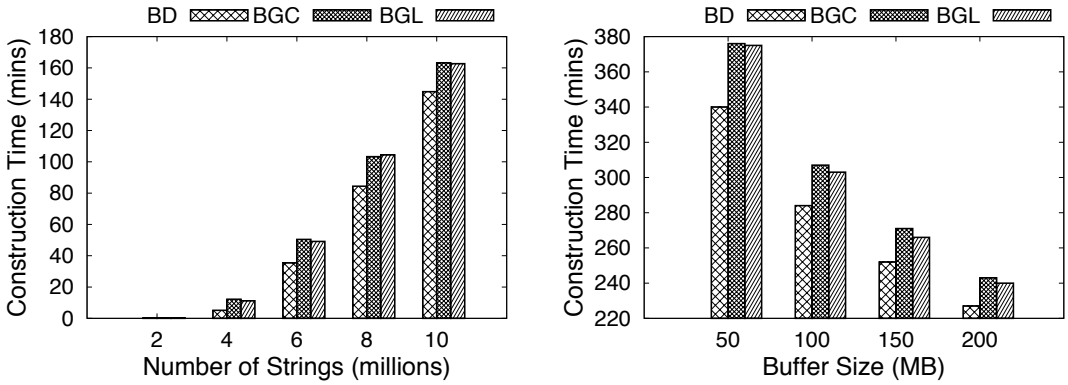
In the left charts of Figures 5.9 and 5.10 we allocated a fixed buffer size of 400MB for index construction. We focused on the the Web Word Grams dataset shown in Figure 5.10 since the results on the Medline dataset had a similar trend. It shows that the index-construction procedure scaled well (almost linearly) with the size of the dataset. The right chart shows the construction performance with varying buffer sizes, and we see that the



(a) 400MB Buffer. (b) 20 million strings.

Figure 5.10: Index-construction performance on Web Word Grams.

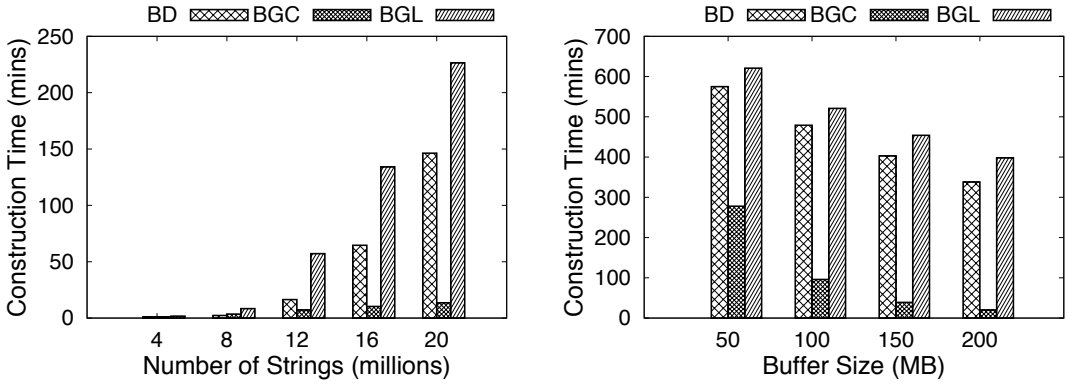
merging of the runs took most of the time. By increasing the buffer size we improved the performance of merging the runs. The other two phases, creating the runs and reorganizing the index, did not benefit from a larger buffer size because they were CPU bound, explained as follows. Creating the runs consists of tokenizing the strings, and frequently reallocating in-memory inverted lists. Reorganizing the index consists of sorting inverted-list elements, while performing disk operations in buffer-sized chunks.



(a) 400MB Buffer. (b) 10 million strings.

Figure 5.11: BED-tree index-construction performance on Medline Titles.

We report the construction times for BED-tree in Figures 5.11 and 5.12 only for completeness. As before, we used a fixed buffer of 400MB in the left charts and varied the buffer size on the right charts for a fixed number of strings. A direct comparison of our construction-



(a) 400MB Buffer.

(b) 20 million strings.

Figure 5.12: BED-tree index construction performance on Web Word Grams.

procedure with BED-tree's would be somewhat unfair since our technique builds its inverted index in "bulk", while BED-tree does not currently implement bulk-loading (it uses repeated insertions).

5.5.2 Query Performance Naming Conventions and Methodology

In this subsection, we introduce the different flavors of BED-tree and the inverted-index approach used in our experiments on query performance. We also detail our procedures for obtaining the results of different types of experiments.

BED-Tree Naming: We follow the convention from [124]. BD, BGC, and BGL refer to a BED-tree using the dictionary order, gram count order, and gram location order, respectively.

Naming of Our Approaches: In our experiments we focused on the following two extreme approaches showing the best and worst inverted-index solutions for raw disk performance. "Simple" refers to a straightforward adoption of existing algorithms. It uses an unpartitioned inverted index and a dense index whose entries are in an arbitrary order. "Simple" retrieves all the inverted lists of a query string's grams and then solves the T-occurrence problem with an efficient in-memory algorithm (we used DivideSkip [76]). We use "AdaptPtOrd" to

refer to our most advanced method using the adaptive algorithm ("Adapt"), a partitioned inverted index ("Pt"), and a dense index with entries ordered by their length ("Ord"). We experimentally explored the various dimensions of our solutions, where we also use letter count filtering indicated by "Lc".

Raw Disk: In this type of experiments, we measured the performance of queries when all data required for answering a query (inverted lists, dense index blocks, BED-tree blocks) needed to be retrieved from disk. To do so, we cleared the filesystem cache before each query. Recall that our inverted-index assumes the FilterTree is in memory (Section 5.3.1). For a fair comparison, we allocated the same amount of memory needed for the FilterTree to BED-tree's buffer manager. Note that BED-tree implements its own buffer manager, and therefore, those blocks cached in its buffer manager were unaffected by clearing the filesystem cache.

We also gathered the number of disk seeks and the amount of data transferred from disk per query captioned as "Data Transferred" and "Disk Seeks". For BED-tree, the disk seeks are the number of nodes retrieved from disk (not already in the buffer manager), and the data transferred is that number multiplied by the block size. For our inverted-index solution, the number of disk seeks is the number of inverted lists and dense-index blocks accessed. We computed the data transferred using the sizes of the inverted lists and dense-index blocks.

Fully Cached Index: This experiment represents the other extreme in which all data required to answer a query is already in memory. For BED-tree we achieved this behavior by allocating a large amount of memory in its buffer manager. We ran our workloads immediately after building the BED-tree, and therefore, the entire BED-tree was in memory when running queries. For our inverted-index approach we relied on the filesystem for caching. We first built the inverted index and dense index without clearing the filesystem cache, and then immediately ran our workloads, assuming that after construction all indexes are probably in the filesystem's cache.

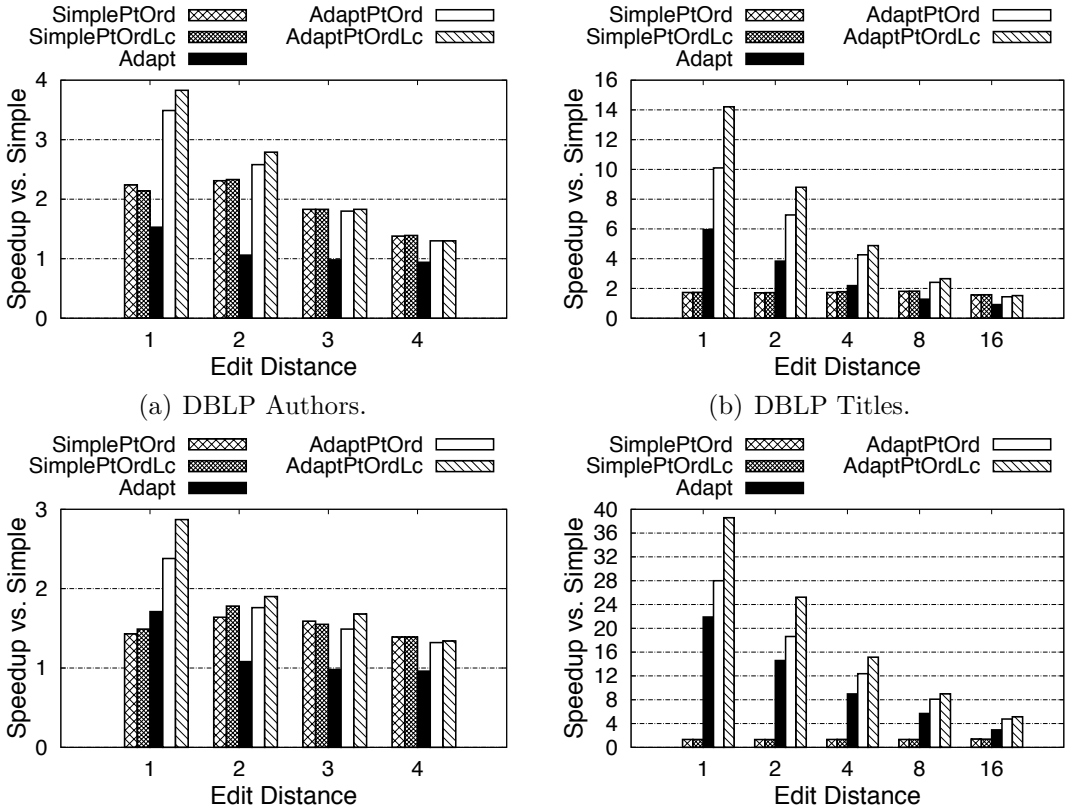
5.5.3 Range Queries Using Edit Distance

5.5.3.1 Improvement Over Simple Inverted-Index Approach

We first examined the relative performances of the inverted-index based approaches, shown in Figures 5.13 and 5.14 for range queries. We plotted the speedups over "Simple" of various combinations of algorithms and indexes for raw disk performance and a fully cached index (a speedup of 1 means no improvement). It is interesting that "Adapt" and "PtOrd" by themselves (i.e., not in combination) delivered moderate speedups, and we achieved the best speedups by combining them, e.g., as "AdaptPtOrd". Also, observe that the letter count filter "Lc" improved the performance even further. The speedups varied so much for different datasets due to the length of their strings and corresponding queries, e.g., the queries on Uniprot had many grams whose inverted lists the adaptive algorithm could avoid reading. It is not surprising that for the fully cached indexes in Figure 5.14, the "Simple" approaches often outperform "Adapt", because "Simple" was designed for in-memory efficiency and "Adapt" trades CPU effort to save I/O effort, and can therefore be slower than "Simple" when the indexes are completely in memory.

5.5.3.2 Comparison with BED-Tree

Next, we compare our approach with BED-tree on range-query performance using the datasets and workloads from the BED-tree paper [124]. When comparing our approach with BED-tree, we restrict our attention to "AdaptPtOrd" and "Simple", to show the best and worst inverted-index approaches for raw disk performance (also assuming we do not wish to spend memory on letter count filtering). The first two graphs, (a) and (b), of Figures 5.15-5.18 show the raw disk, and fully cached index times, respectively. The graphs (c) and (d) further detail the raw disk performance with the average number of disk seeks and



(a) DBLP Authors.

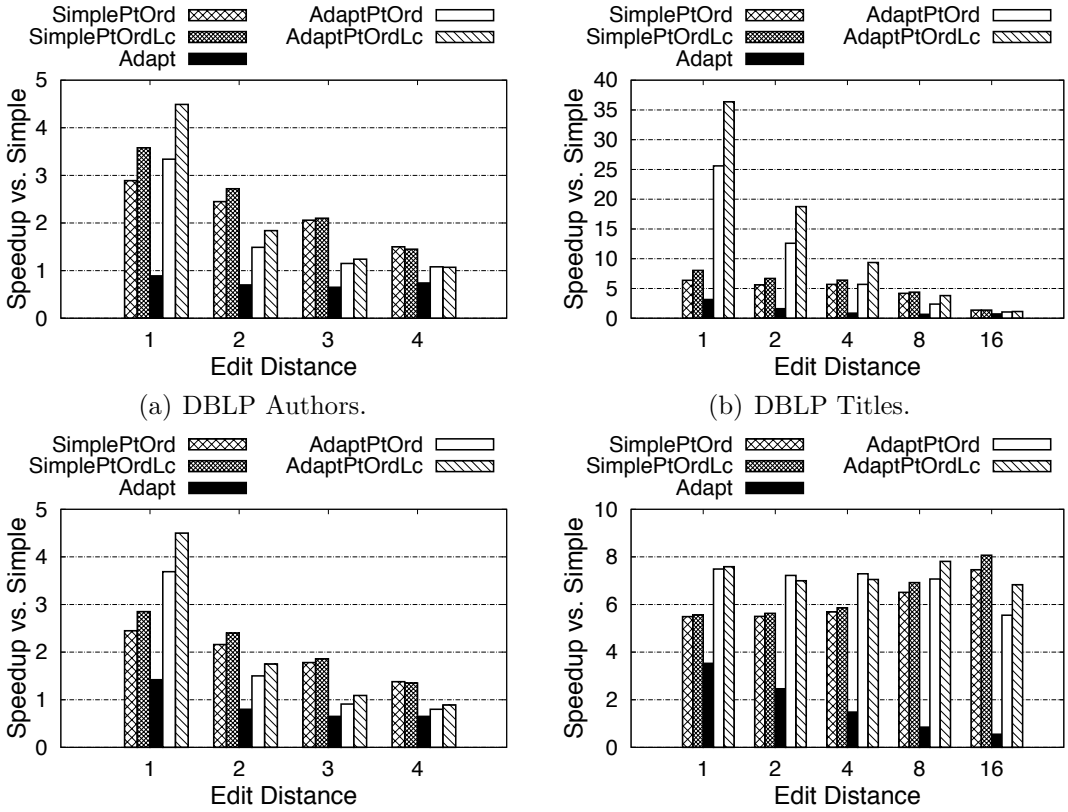
(b) DBLP Titles.

(c) IMDB Actors.

(d) Uniprot.

Figure 5.13: Speedup of range queries over the Simple approach using raw disk.

data transferred per query. We see that "AdaptPtOrd" consistently outperformed BED-tree by orders of magnitude, both for raw disk peformance and for a fully cached index (notice we are using a log scale on the y-axes). Also, "AdaptPtOrd" was considerably faster than "Simple". The performance differences between "AdaptPtOrd" and the BED-tree variants on raw disk are explained by the graphs (c) and (d). "AdaptPtOrd" transferred significantly less data per query with fewer disk seeks than BED-tree. The two main reasons why BED-tree examined so many nodes are as follows. First, the pruning power at higher levels of the BED-tree is weak because a node entry refers to the enclosing interval of ranges in its subtree. Second, the BED-tree search procedure traverses multiple paths in the tree (more akin to an R-tree search [45]), leading to additional node accesses as compared to a standard B-tree range search. Such a search procedure can also incur long disk-seek distances,



(a) DBLP Authors.

(b) DBLP Titles.

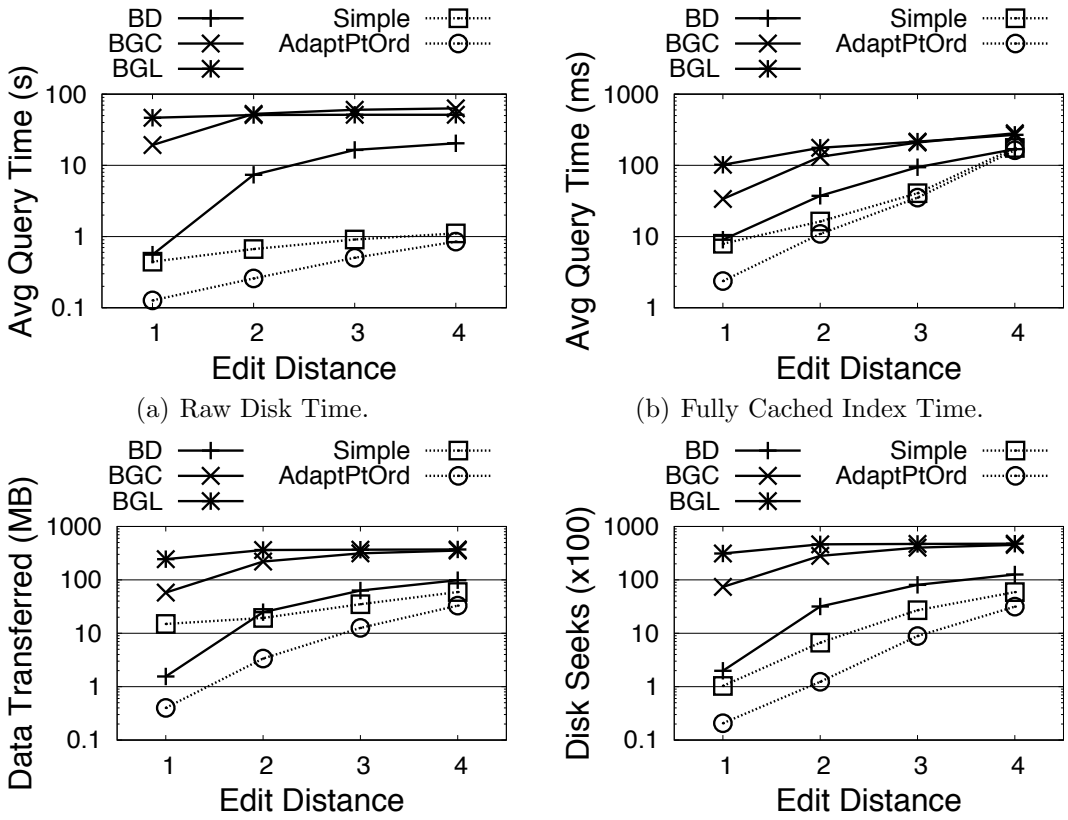
(c) IMDB Actors.

(d) Uniprot.

Figure 5.14: Speedup of range queries over the Simple approach using a fully cached index.

because it is impossible to simultaneously store all tree-nodes close to each other (whereas in a standard B-tree we only need to store the leaves close to their siblings). Our argument that BED-tree's pruning power is not as strong as our approach is also supported by the results on fully cached indexes, where a significant cost is computing the real edit distances to candidate answers.

In Table 5.2 we summarize the index sizes for these experiments. We observe that, in general, our inverted index approach requires more space than BED-tree. For example, on the DBLP Authors dataset, the BED-tree with dictionary ordering (BD) required 97MB of disk-space, and our indexes required 82 + 204 = 286MB on disk. However, our approaches transferred much less data from disk per query (see Figures 5.15-5.18). For the raw disk experiments we give the BED-tree variants a buffer space equal to the size of our FilterTree ("FT").



(a) Raw Disk Time.

(b) Fully Cached Index Time.

(c) Data Transferred.

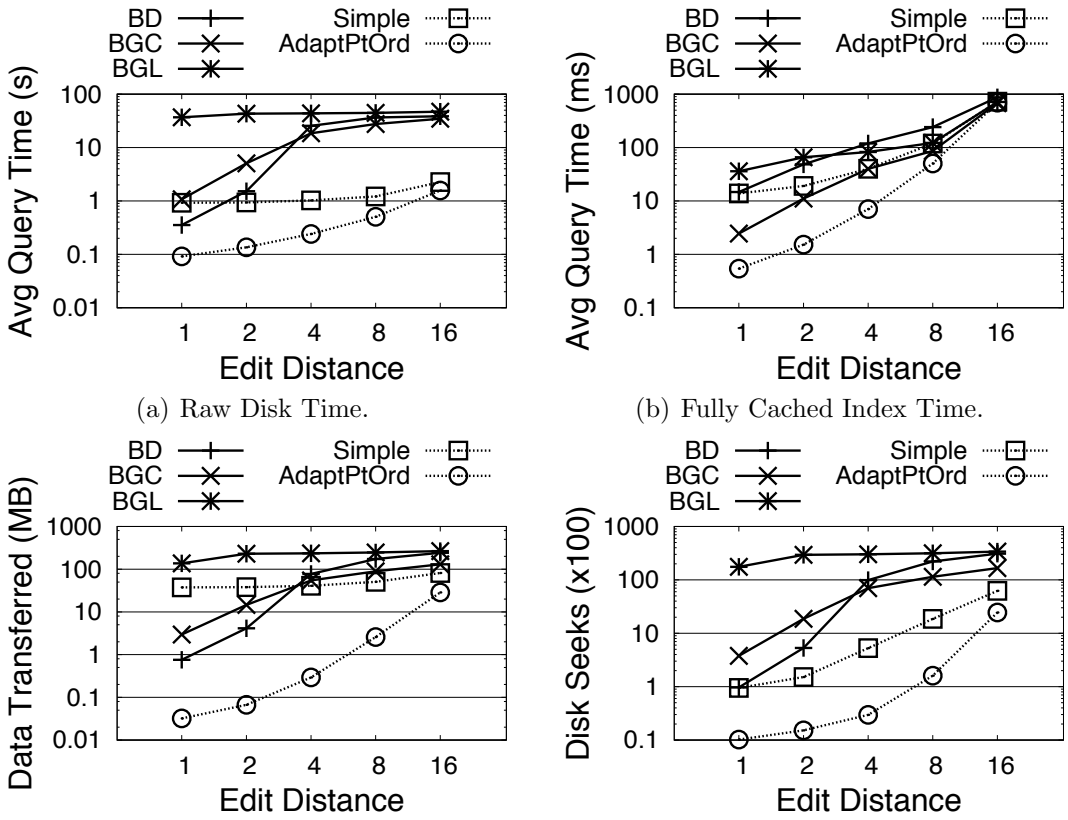
(d) Disk Seeks.

Figure 5.15: Range-query performance on DBLP Authors.

5.5.3.3 Scalability

In Figures 5.19 and 5.20 we varied the number of indexed strings on our two large datasets, Web Word Grams and Medline Titles, to evaluate the scalability of our techniques. Due to its slow performance we omit BED-tree from the raw disk experiments. For example, on 12 million Web Word Grams, its best version (BD) needed an average of 15 seconds per query, and on 6 million Medline Titles its best version (BGC) needed an average of 145 seconds. Similarly, we only plot the best version of BED-tree for the in-memory results since the other versions were significantly worse.

Our results show that "AdaptPtOrd" offers better scalability than "Simple", explained as follows. As we increased the size of the dataset, some inverted lists became longer. However,



(a) Raw Disk Time.

(b) Fully Cached Index Time.

(c) Data Transferred.

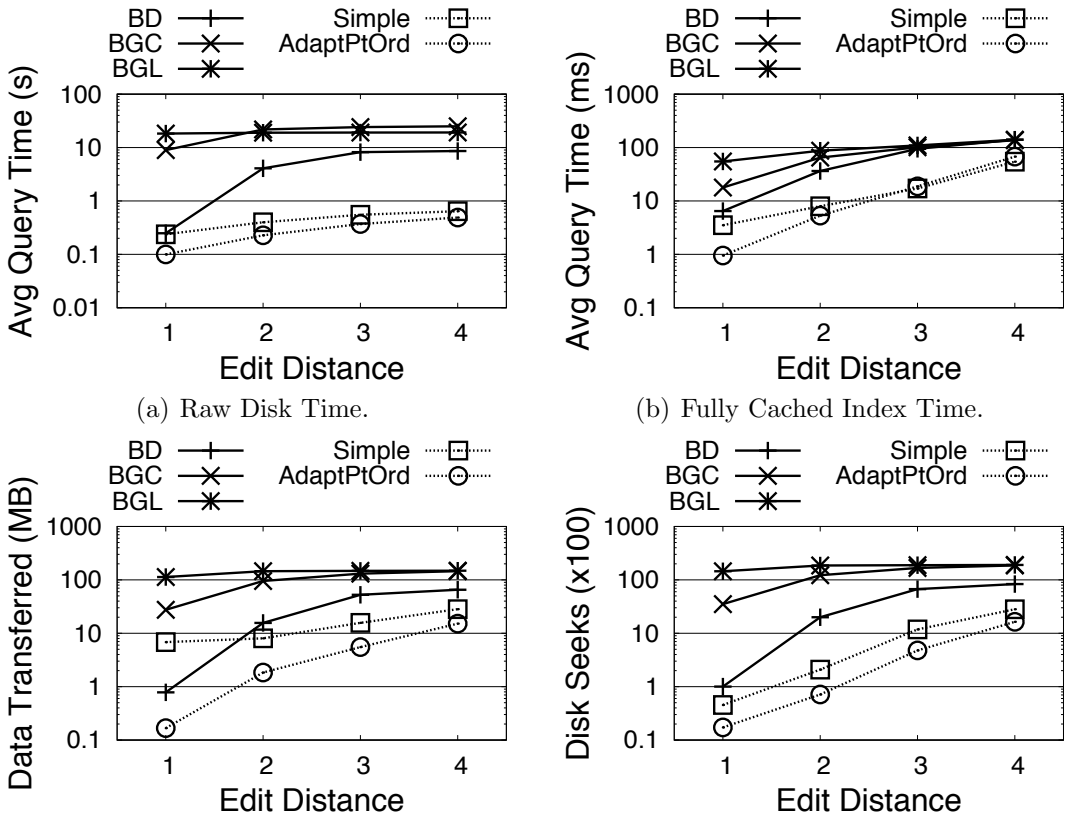
(d) Disk Seeks.

Figure 5.16: Range-query performance on DBLP Titles.

the number of results per query grew relatively slower than the total index size. Especially for highly selective queries, the adaptive algorithm avoided processing many unnecessary inverted lists. This effect explains the excellent performance on the highly selective Medline Titles. Similar arguments hold for the experiments with fully cached indexes.

5.5.4 Top-K Queries Using Edit Distance

Next, we show the results on top-k queries. As before we used the datasets and workloads of the BED-tree paper. We answered top-k queries on our inverted index by a series of range



(a) Raw Disk Time.

(b) Fully Cached Index Time.

(c) Data Transferred.

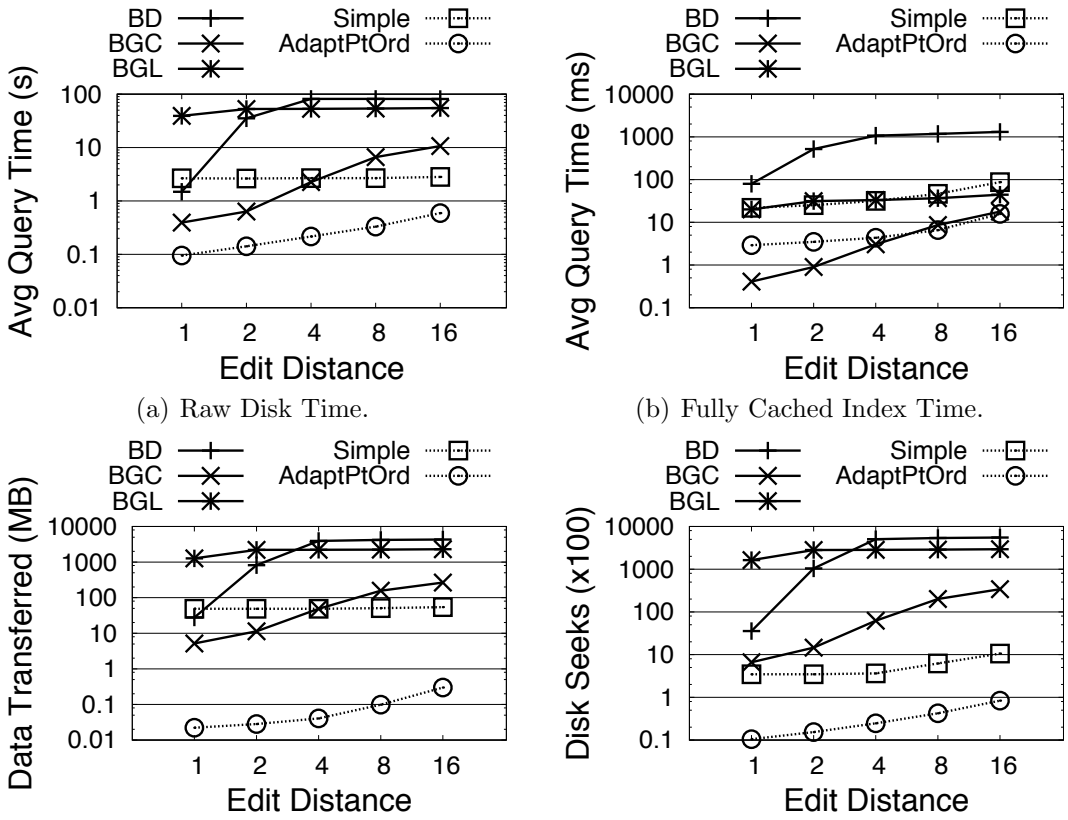
(d) Disk Seeks.

Figure 5.17: Range-query performance on IMDB Authors.

queries with increasing edit-distance thresholds based on the following equation.

$$ed_{n+1}(s) = ed_n(s) + max(\lfloor\sqrt{ed_n + \frac{|s|}{b}}\rfloor, 1) \tag{5.6}$$

In Equation 5.6 $ed_n(s)$ denotes the n-th edit-distance threshold used in a range query as part of answering a top-k query with string s. We created the equation by running a few queries (not part of the tested workloads) and examining their behavior. Intuitively, the equation reflects that for longer strings we should increase the edit-distance threshold faster than for shorter strings ($\lfloor\sqrt{ed_n + \frac{|s|}{b}}\rfloor$). The number b is a tuning parameter to dampen the effect of the string-length $|s|$ on the edit-distance growth rate. We set $b = 10$ in our experiments.



(a) Raw Disk Time.

(b) Fully Cached Index Time.

(c) Data Transferred.

(d) Disk Seeks.

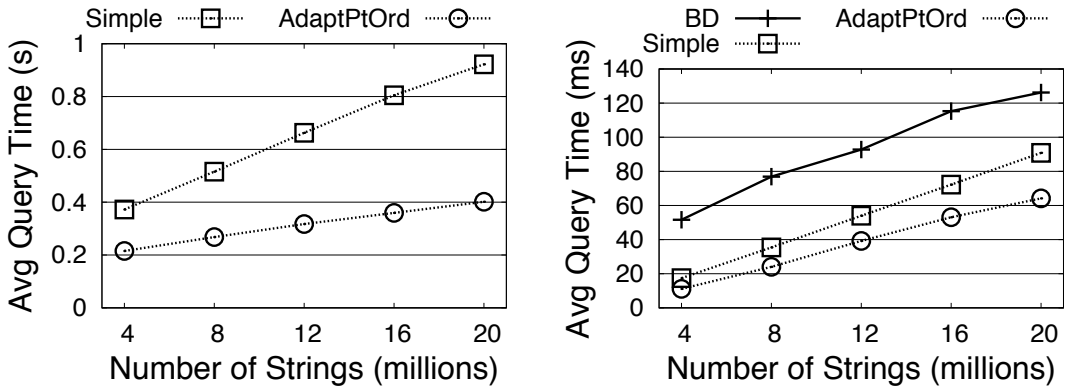
Figure 5.18: Range-query performance on Uniprot.

When answering top-k queries we start with $ed_1 = 0$ to find all exact matches. If we have not yet found at least k answers then we run another range query with a higher edit-distance threshold, ed_2, computed using Equation 5.6, and take the union of the previous results and those new results. We continue this process as long as we have fewer than k distinct answers. Note that, eventually, we might revert to a full scan of the dataset if the edit-distance threshold increases to a point where the T lower bound becomes zero or negative (see Section 2.4.1).

We do not show results for the DBLP Titles and Uniprot datasets because for some queries BED-tree did not find all correct answers. For example, on Uniprot with $K = 4$, BED-tree only returned a total of 304 answers instead of the correct 400 answers for the 100 queries.

Dataset	BD	BGC	BGL	DenIx	InvIx	FT
DBLP Author	97	225	189	82	204	7
DBLP Title	123	156	157	100	297	21
IMDB Actor	38	88	75	32	88	11
Uniprot	283	302	305	222	617	49

Table 5.2: Index sizes in MB. Sizes of BED-tree variants and our inverted-index components. DenIx refers to the dense index, InvIx to the inverted index, and FT to the in-memory FilterTree.



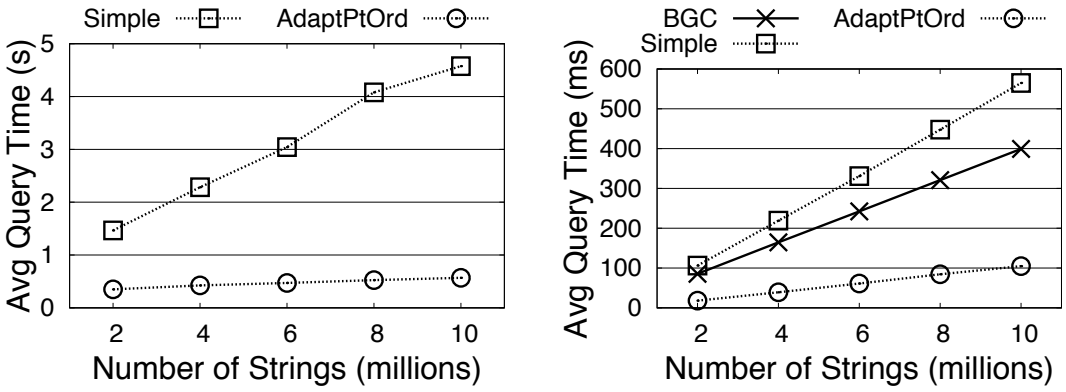
(a) Raw Disk Time.

(b) Fully Cached Index Time.

Figure 5.19: Range-query scalability with edit distance 2 on Web Word Grams.

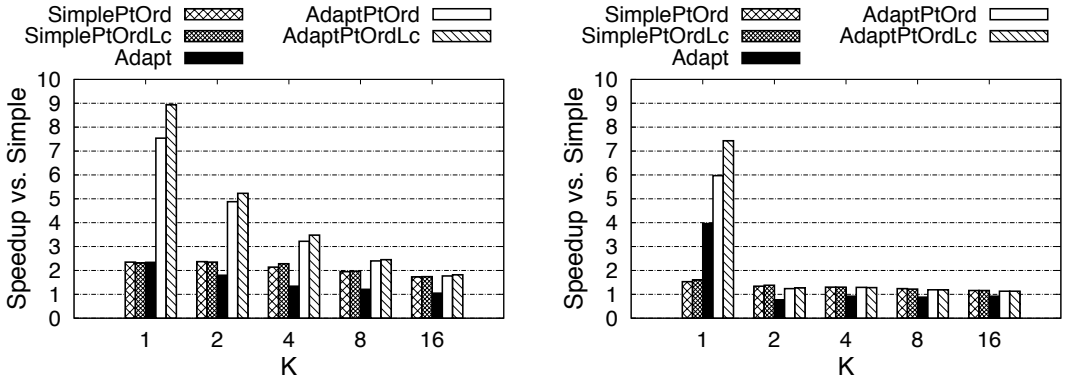
5.5.4.1 Improvement Over Simple Inverted-Index Approach

Figures 5.22 and 5.22 show the relative performances of our approaches on top-k queries. The reason why the speedups for top-1 queries are so high is that our query workload consisted of strings taken from the dataset. Therefore, a top-1 query is essentially an exact-match query, but it is nevertheless interesting to note that our techniques can efficiently answer exact-match queries as well. In general, we observed moderate speedups (mostly between 1 and 3) for top-k queries both on raw disk performance and on fully cached indexes. The reason why the speedups were only moderate is that "Simple" already achieved a reasonable performance. The difference between "Simple" and "AdaptPtOrd" is presented more clearly in Figures 5.23 and 5.24 of the following subsection.



(a) Raw Disk Time. (b) Fully Cached Index Time.

Figure 5.20: Range-query scalability with edit distance 6 on Medline Titles.

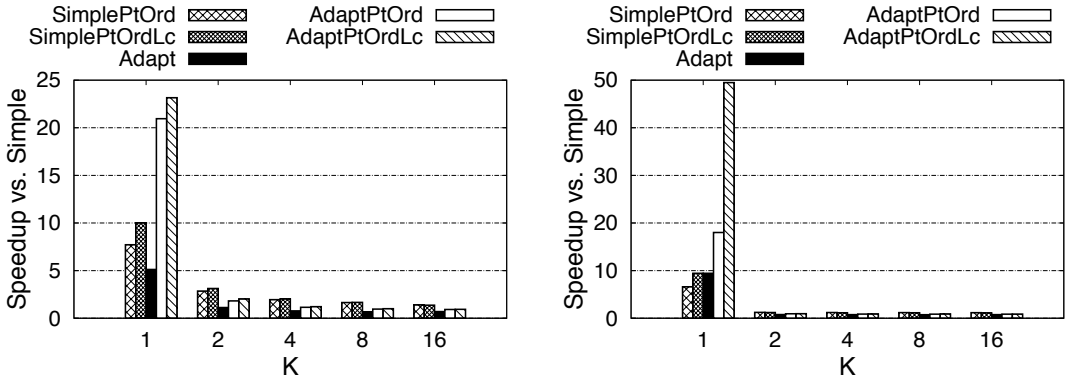


(a) DBLP Authors. (b) IMDB Actors.

Figure 5.21: Speedup of top-k queries over the Simple approach using raw disk.

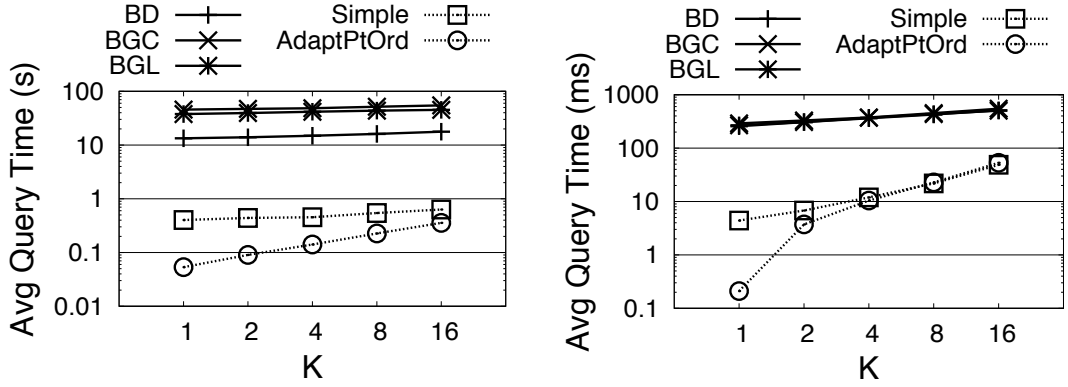
5.5.4.2 Comparison with BED-Tree

The results on top-k queries shown in Figures 5.23 and 5.24 are consistent with those on range queries, and they showed that that our techniques answered top-k queries efficiently and outperformed BED-tree.



(a) DBLP Authors. (b) IMDB Actors.

Figure 5.22: Speedup of top-k queries over the Simple approach using a fully cached index.



(a) Raw Disk Time. (b) Fully Cached Index Time.

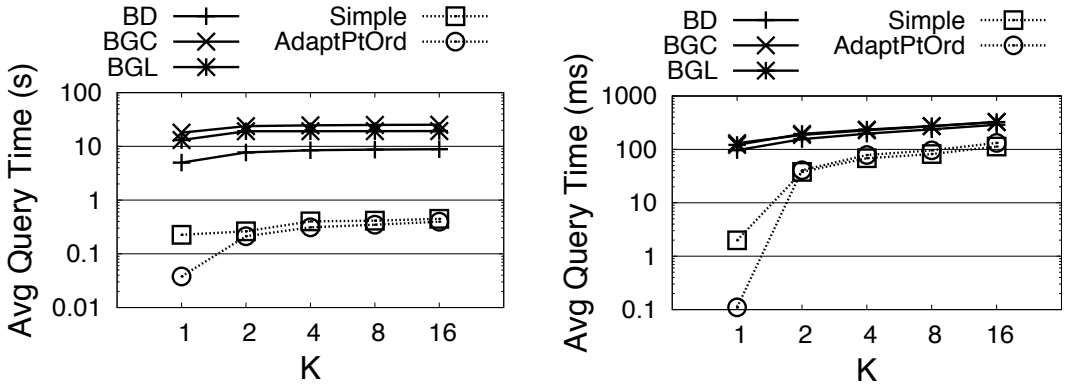
Figure 5.23: Top-K query performance on DBLP Authors.

5.5.5 Normalized Edit Distance

In this subsection we present experiments on range queries using normalized edit distance with the datasets and workloads from the BED-tree paper. As before, we first present the relative performances of our approaches and then compare them with BED-tree.

5.5.5.1 Improvement Over Simple Inverted-Index Approach

Figures 5.26 and 5.26 show the relative performances of our approaches. We did not implement letter count filtering ("Lc") for normalized edit distance, and therefore we only show



(a) Raw Disk Time.

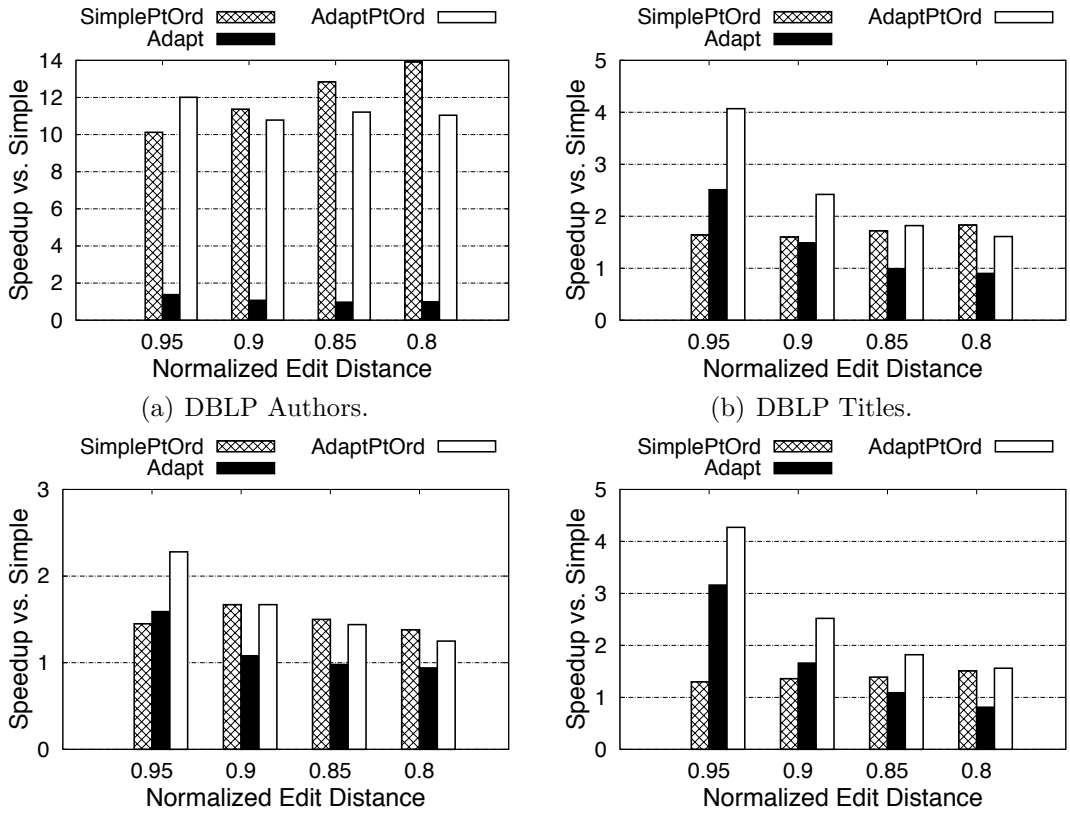
(b) Fully Cached Index Time.

Figure 5.24: Top-K query performance on IMDB Actors.

the speedups of "SimplePtOrd", "Adapt", and "AdaptPtOrd" versus "Simple". We see that our approaches offered a good improvement over "Simple". Similar to the results on edit distance, our results on normalized edit distance showed that "AdaptPtOrd" mostly performed best for raw disk performance, and "SimplePtOrd" performed best on a fully cached index. As before, the reason is that "AdaptPtOrd" spends additional CPU cycles to save I/O costs, whereas "SimplePtOrd" is designed for efficient in-memory operation. In systems with explicit control on caching data from disk, one could easily switch from the adaptive algorithm to an in-memory algorithm (such as DivideSkip) when most inverted lists for a particular query are already in memory. In this fashion, we could ensure good raw disk performance and good fully cached index performance (note that both algorithms can operate on the same index structure).

5.5.5.2 Comparison with BED-Tree

Since BED-tree currently supports normalized edit distance only with the gram counting order, we have only BGC in Figures 5.27 and 5.28. As before, "AdaptPtOrd" outperformed its competitors.



(a) DBLP Authors.

(b) DBLP Titles.

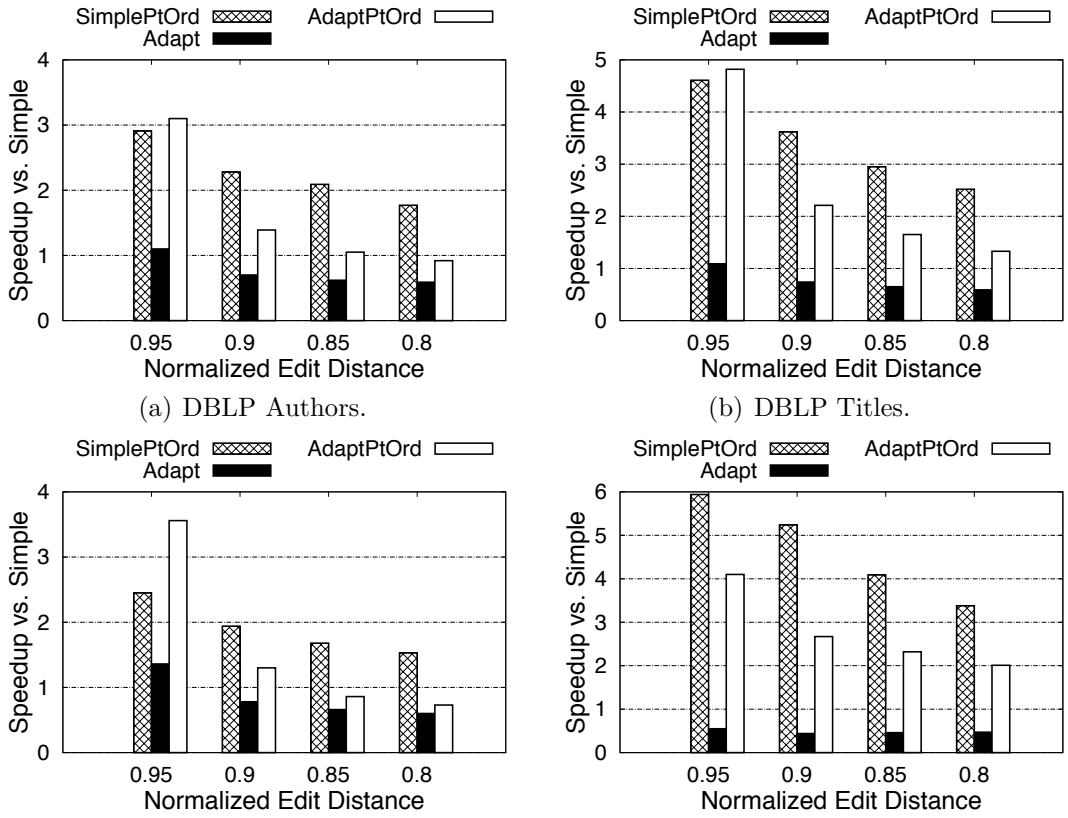
(c) IMDB Actors.

(d) Uniprot.

Figure 5.25: Speedup or range queries over the Simple approach using raw disk.

5.5.6 Jaccard with Q-Gram Tokens

In Figures 5.31 and 5.32 we used the Jaccard similarity of multisets of q-gram tokens to quantify the similarity between strings. Though BED-tree could possibly answer queries using Jaccard with the gram count ordering, its current implementation does not support it. Therefore, we only plot the results of our approaches. We observe that our new techniques also provide a benefit to queries using Jaccard.



(a) DBLP Authors.

(b) DBLP Titles.

(c) IMDB Actors.

(d) Uniprot.

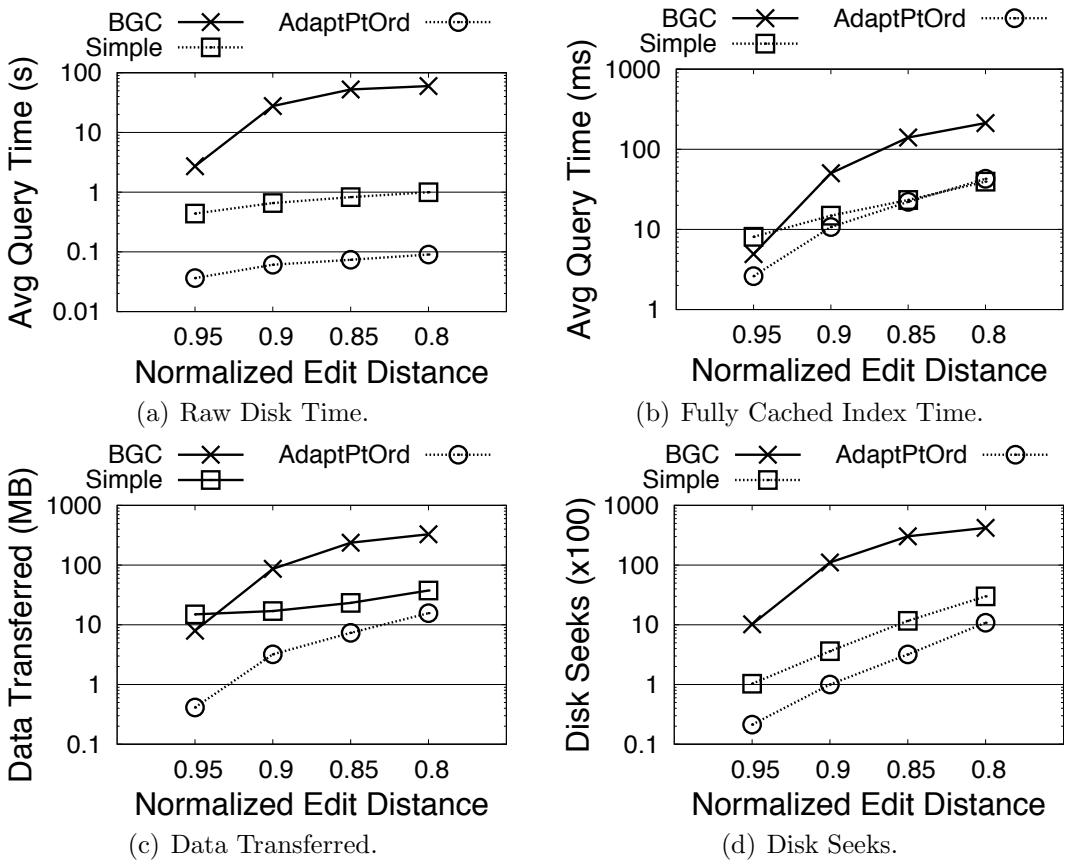
Figure 5.26: Speedup or range queries over the Simple approach using a fully cached index.

5.5.7 Jaccard with Word Tokens

For those datasets with very long strings (DBLP Titles and Medline Titles), it could be more meaningful to use Jaccard based on word tokens to quantify the similarity between strings. Figures 5.33 and 5.34 show the raw disk performance of queries based on word tokens, and we see that our techniques also improved their performance in this setting.

5.6 Conclusion

We have studied global-alignment approximate string selection queries when the data and indexes reside on disk. To support such queries, we proposed a new physical layout for an



(a) Raw Disk Time.

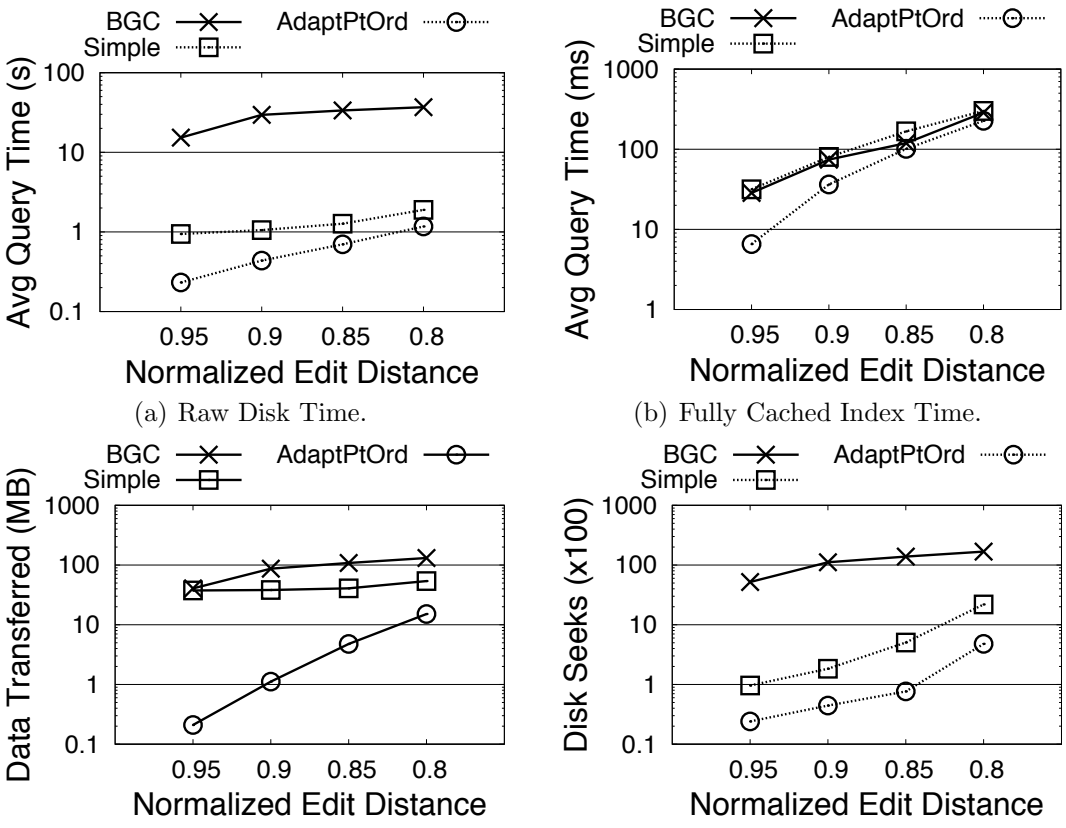
(b) Fully Cached Index Time.

(c) Data Transferred.

(d) Disk Seeks.

Figure 5.27: Range-query performance on DBLP Authors.

inverted index, demonstrated how to efficiently construct it, and showed its benefits to query processing. We developed a cost-based adaptive algorithm to answer queries; it balances the costs of retrieving inverted lists and candidate answers from disk. We have shown that the adaptive algorithm and the new index layout complement each other and that their combination answers queries efficiently. Further, our techniques significantly outperformed a recent tree-based index, BED-tree.



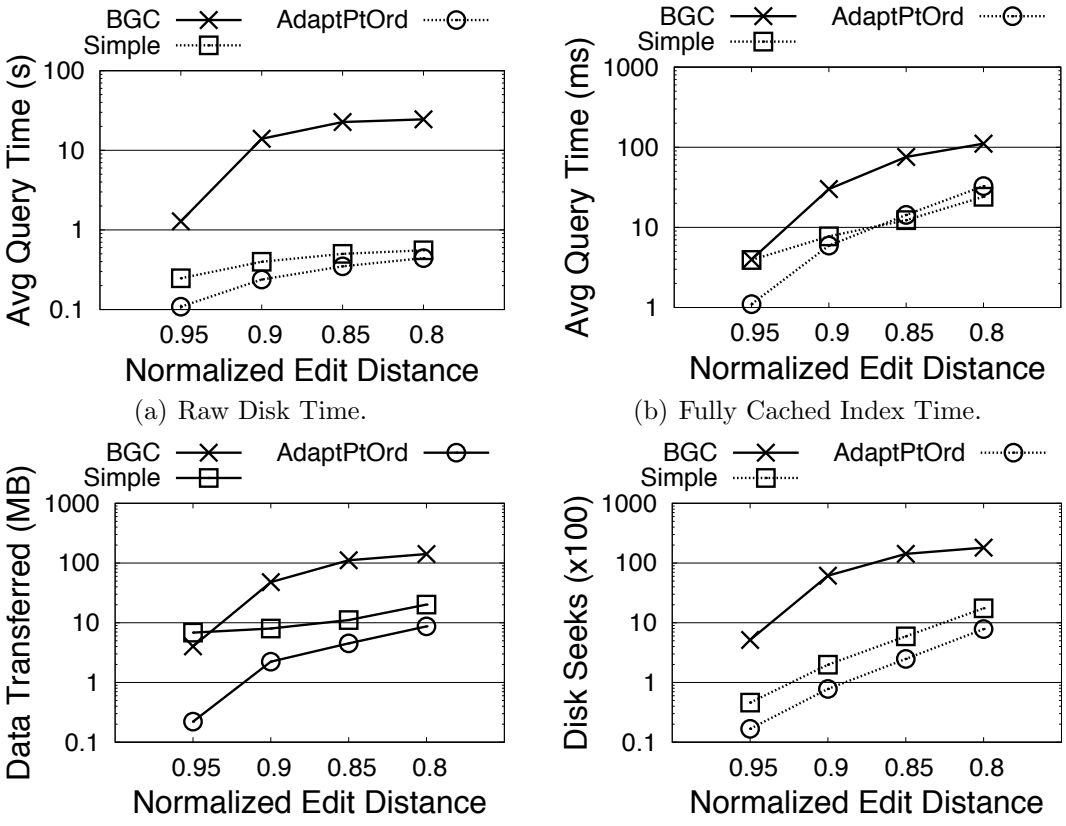
(a) Raw Disk Time.

(b) Fully Cached Index Time.

(c) Data Transferred.

(d) Disk Seeks.

Figure 5.28: Range-query performance on DBLP Titles.



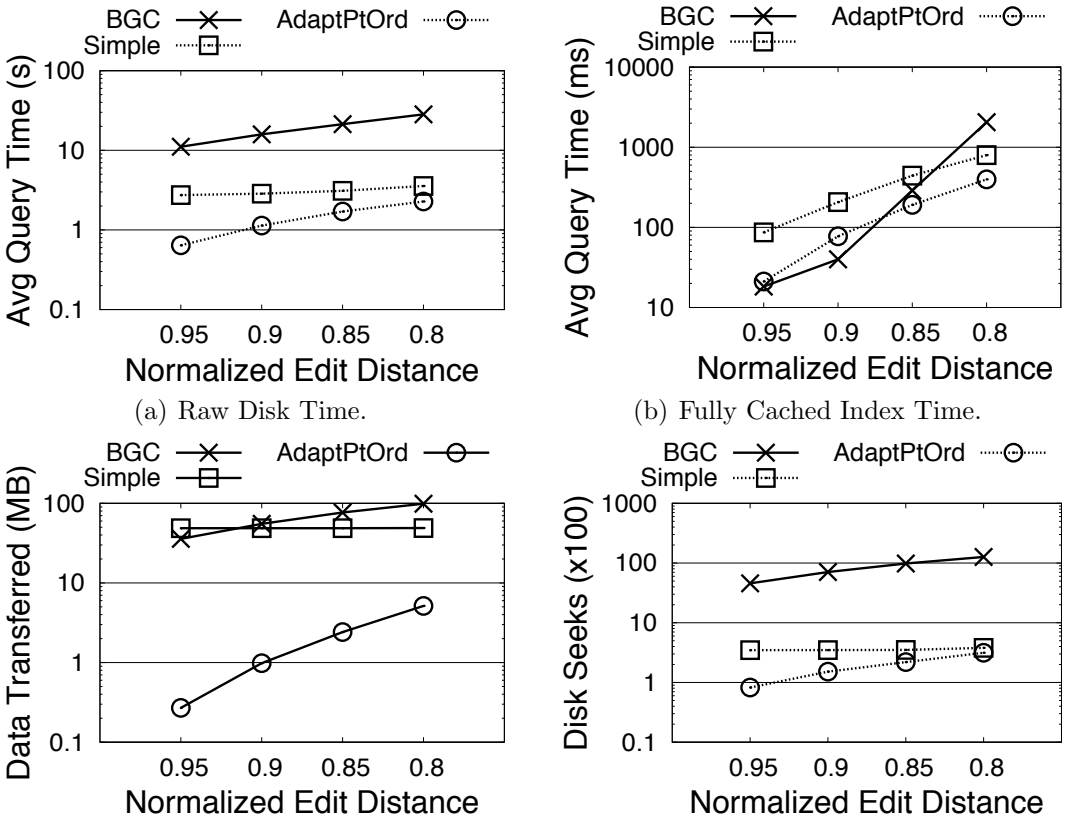
(a) Raw Disk Time.

(b) Fully Cached Index Time.

(c) Data Transferred.

(d) Disk Seeks.

Figure 5.29: Range-query performance on IMDB Authors.



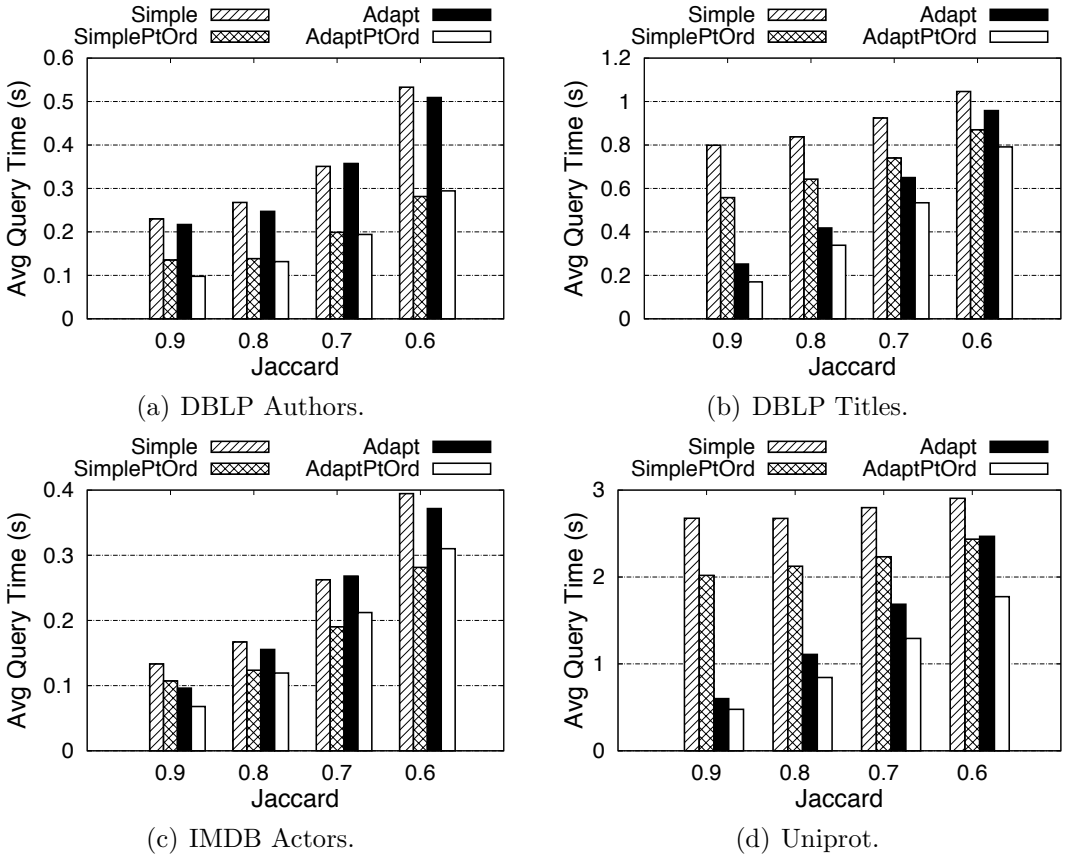
(a) Raw Disk Time.

(b) Fully Cached Index Time.

(c) Data Transferred.

(d) Disk Seeks.

Figure 5.30: Range-query performance on Uniprot.



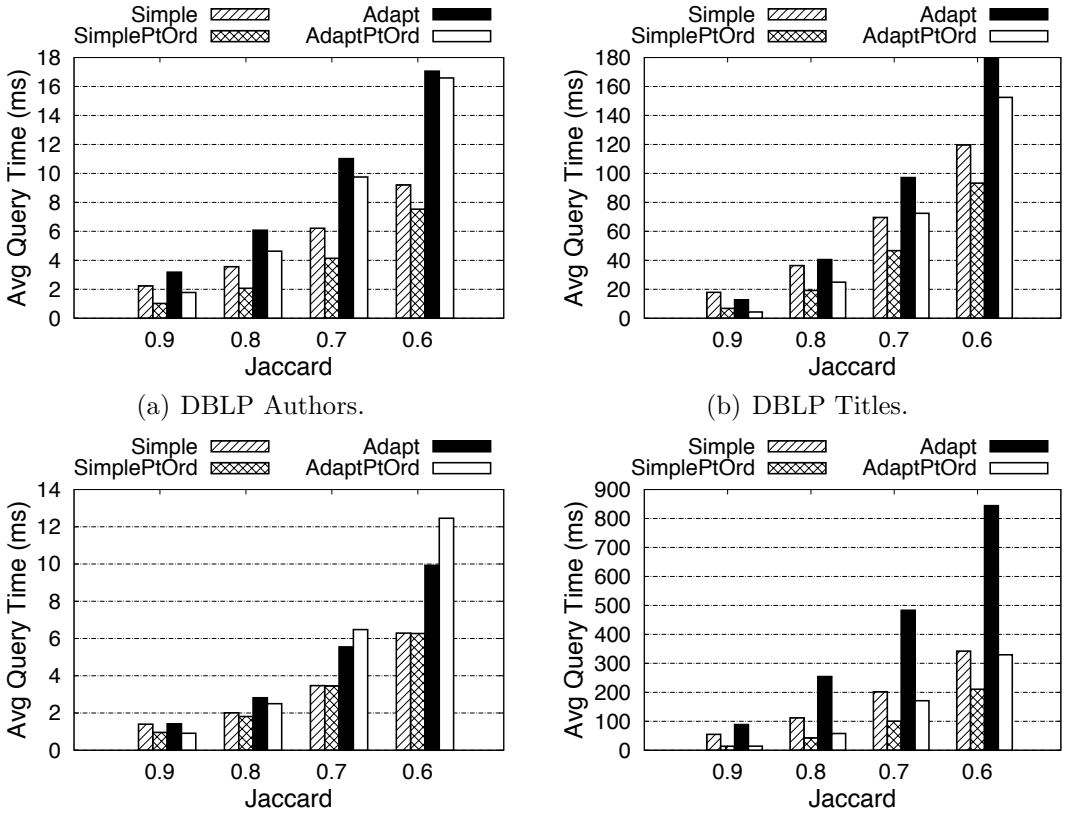
(a) DBLP Authors.

(b) DBLP Titles.

(c) IMDB Actors.

(d) Uniprot.

Figure 5.31: Raw disk range-query performance using Jaccard on q-gram tokens.



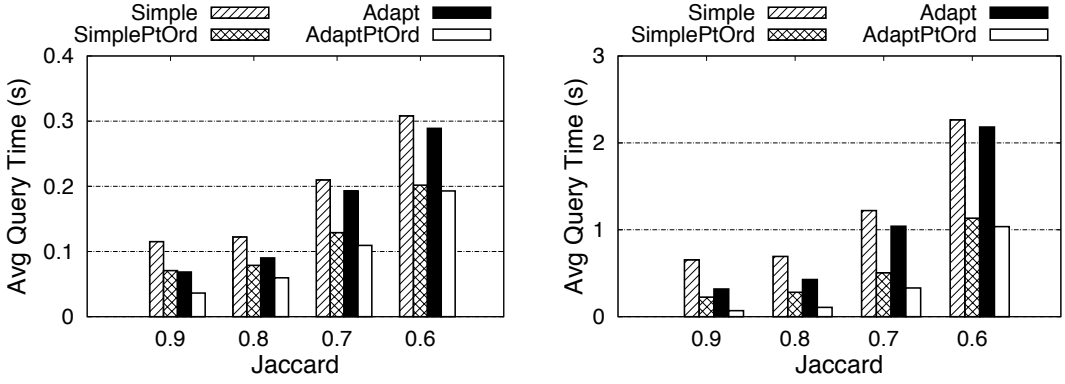
(a) DBLP Authors.

(b) DBLP Titles.

(c) IMDB Actors.

(d) Uniprot.

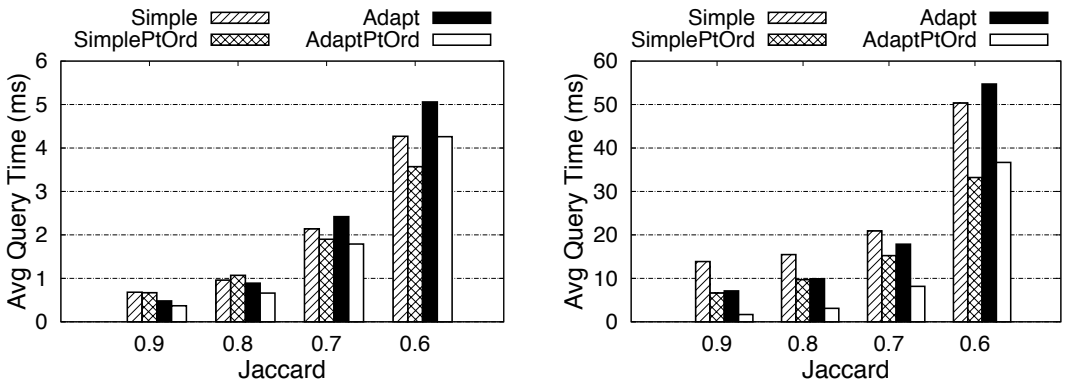
Figure 5.32: Fully cached index range-query performance using Jaccard on q-gram tokens.



(a) DBLP Titles.

(b) Medline Titles.

Figure 5.33: Raw disk range-query performance using Jaccard on word tokens.



(a) DBLP Titles.

(b) Medline Titles.

Figure 5.34: Fully cached index range-query performance using Jaccard on word tokens.

Chapter 6

Indexed Set-Similarity Queries in ASTERIX

In this chapter, we present the implementation of indexed support for set-similarity selection and join queries in the ASTERIX parallel database system. We discuss how set-similarity queries are expressed in the declarative ASTERIX Query Language (AQL), our implementation of indexes capable of optimizing such queries, and finally the rewriting of set-similarity selection and join queries using indexes. We conclude with an experimental evaluation of different index types and query-plan alternatives, as well as a comparison against an existing parallel set-similarity join technique [110].

6.1 Introduction

With set-similarity queries being useful in many different applications, we studied how to add such querying capabilities to our general-purpose database called ASTERIX, hoping to benefit a wide variety of uses cases. Further, we wish to enable application develop-

ers to conveniently express set-similarity queries in a high-level declarative query language. ASTERIX [14, 17] is a shared-nothing, parallel database management system for storing, indexing, and querying large quantities of semi-structured data. It is geared towards the new breed of "Big Data" applications which aim to derive knowledge from the vast quantities of data being generated on a daily basis on the Web, for example, in blogs, social networks, online communities, click streams, etc. Data gathered from these sources is bound to contain noise and inconsistencies, since many users publish their data informally resulting in abbreviated keywords and misspellings, e.g., `Obama inauguration` vs. `Obama inaguration` and `Bama` vs. `Alabama`. In addition, analyzing such data, e.g., to make recommendations or to identify sub-populations and trends in social networks, often requires the matching of multiple sets of data based on set-similarity measures such as Jaccard. For example, suppose a news provider wants to optimize its Web page layout by analyzing the success of past news stories. Apart from article-specific measures, such an analysis could take into account the general impact of news stories on certain combinations of topics on the Twitter community, i.e., we want count how many Tweets are "related" to a particular news story based on the similarity of their mentioned topics. The scale of the data we are targeting motivates us to provide indexed support for answering such queries. In that context, we experiment with different plan alternatives when applying indexes to set-similarity selection and join queries to understand their performance tradeoffs. To that end, we compare index-based set-similarity joins with an existing non-indexed parallel method for executing such joins [110]. Our goal is to gain an initial understanding of when our query optimizer should prefer one plan over the other, and to guide users in deciding whether secondary indexes are worthwhile for their workload.

6.1.1 ASTERIX Software Layers

The ASTERIX system encompasses three separate software layers, from top to bottom: ASTERIX, Algebricks, and Hyracks.

ASTERIX Layer: At the top sits the ASTERIX layer that implements the ASTERIX Data Model, (ADM) which can express a superset of JSON (stored in binary), and the ASTERIX Query Language (AQL), an XQuery-like declarative query language. Data in ASTERIX is stored in *datasets* that contain data instances conforming to an *ADM data type*. A dataset is roughly equivalent to a "table" in relational databases. An ADM type specifies the "schema" of a dataset. An ADM type can contain any number of named fields of a primitive (such as integer, string) or complex data type (such as list, set, or ADM record type), and may have arbitrary levels of nesting. An ASTERIX *dataverse* contains many ASTERIX types and datasets, and is analogous to a "database" in relational systems. The ASTERIX layer is also responsible for storing metadata on said types and datasets, including information on partitioning and on primary and secondary indexes. ASTERIX accepts queries in the ASTERIX Query Language (AQL). Such queries are parsed and analyzed by ASTERIX to produce an initial logical query plan. The logical query plan is then fed into Algebricks for rewriting.

Algebricks Layer: Algebricks is a data-model and query-language agnostic query rewriter. It implements a rule-based query optimizer including common logical rewrites such as pushing selections and projections, join inference, sub-query decorrelation, etc. Algebricks also performs physical rewriting by reasoning about physical properties such as partitioning, e.g., for minimizing data redistribution, or for multi-level parallel aggregation. Algebricks presents hooks intended to be implemented by specific query languages such as ASTERIX to provide Algebricks with system-specific metadata such as available access methods. After performing logical and physical optimization, Algebricks produces a final execution plan (a

Hyracks job) for our runtime platform called Hyracks [20].

Hyracks Layer: Hyracks is a generic data-parallel execution environment that resembles those of other parallel databases. At the Hyracks level, data is represented as tuples of bytes making it data-model agnostic. It implements several common operators such as joins, groupby, sorting, data exchanges, and others in such a data-model agnostic fashion. For example, a Hyracks Hybrid Hash Join operator must be provided with appropriate hashing and comparison functions, which are then applied to one or more of the Hyracks tuples' fields (as provided during operator instantiation). In a similar way, Hyracks also implements basic storage facilities such as B-Trees that interact with storage devices through a Hyracks BufferCache. Most recently, we have been developing a Log-Structured-Merge (LSM) framework for write-optimizing "standard" indexes inspired by the LSM-Tree technique [100]. We will shortly explain the LSM concept in more detail, as necessary to understand the implementation of our indexes for set-similarity queries. The ASTERIX native storage system is built on top of Hyracks's LSM framework.

6.1.2 ASTERIX Secondary Indexes

In ASTERIX all data items in native datasets must be uniquely identified by a logical key (primary key). ASTERIX datasets are horizontally hash-partitioned across all nodes on their primary key, and on each node the data items are stored in LSM B-Trees (primary index) residing on locally attached storage devices. The primary index provides efficient retrieval of data items via their primary key. ASTERIX also supports *local* secondary indexes. A local secondary index is co-partitioned with its corresponding primary index, i.e., a secondary-index partition on some node only refers to data items residing in the corresponding primary-index partition on that same node. Secondary indexes in ASTERIX map from their secondary-index keys to the corresponding primary keys (as opposed to mapping to a

physical identifier like a record id). Answering queries using local secondary indexes typically involves probing all the secondary index partitions followed by looking up the resulting primary keys in the corresponding co-located primary index partitions. In this chapter, we focus on secondary inverted indexes for supporting set-similarity queries, and we will shortly discuss the execution of such queries using indexes in more detail.

6.2 User Model: ASTERIX Query Language

In this section, we introduce the ASTERIX Data Model (ADM) and the ASTERIX Query Language via focused examples. We will use the ASTERIX Data Definition Language (DDL) to set up a dataverse and dataset as our running example (analogous to a database and a table in relational systems). We will present several equivalent ways to express similarity selection and join queries in AQL, and explain how to create supporting secondary indexes.

6.2.1 Running Example

To illustrate the support for set-similarity queries in ASTERIX, we use a simple dataset of Facebook users. Figure 6.1 shows the DDL to create such a dataset.

We first create a new dataverse called "SocialData", and use the dataverse. Using a dataverse is similar to connecting to a database in other systems, and it implies that all subsequent AQL statements are executed in the context of that dataverse. Next, we create the type "FacebookUserType" to model a Facebook user. Each Facebook-user instance has a unique id, a screen name, a user's last status message, a date when that user joined Facebook, a field for storing an unordered list of friend ids, and an ordered list of interest ids. We assume that each interest is assigned an id elsewhere, and each Facebook user only stores a list of those interest ids. The qualifier "as open" in the type definition signifies that a particular

```
create dataverse SocialData;
use dataverse SocialData;

create type FacebookUserType as open {
 id: int32,
 name: string,
 last-status-msg: string,
 user-since: datetime,
 friend-ids: {{int32}},
 interests: [int32]
}

create dataset FacebookUsers(FacebookUserType) partitioned by key id;
```

Figure 6.1: Data Definition Language to create a dataset of Facebook users.

Facebook user may have any number of fields, in addition to those already declared in the type. Finally, we create a dataset "FacebookUsers" that stores data instances conforming to the "FacebookUserType" data type. We specify the "id" field as the primary key and partitioning key.

Now that we have created a dataverse, a data type, and a dataset, we may insert data instances into the "FacebookUsers" dataset, as shown in Figure 6.2. The example shows two insert statements that each add a single data instance. The data instances are created by invoking the record constructor function "{ }". Similarly, the data values for complex fields are also created via their appropriate constructor functions, e.g, "datetime()" for the user-since field, "{{ }}" for an unordered list of friend ids, and "[ ]" for the ordered list of interests. Notice that the data instance in the second insert statement has an additional field "pet-name" which was not declared as part of the FacebookUserType. Such additional fields are accepted for types declared as open.

```
insert into dataset FacebookUsers {
 id: 1,
 name: "Sir Patrick Stewart",
 last-status-msg: "Merry Christmas & a Happy New Year!",
 user-since: datetime("2007-01-01T00:00:00Z"),
 friend-ids: {{4, 6, 100, 67, 34, 2}},
 interests: [1, 43, 11, 23]
}

insert into dataset FacebookUsers {
 id: 100,
 name: "Brent Spiner",
 last-status-msg: "Having breakfast pancakes, so good.",
 user-since: datetime("2008-01-01T00:00:00Z"),
 friend-ids: {{5, 11, 342, 149, 12, 2}},
 interests: [3, 23, 162, 50],
 pet-name: "Spot"
}
```

Figure 6.2: Inserting two example instances into the "FacebookUsers" dataset.

6.2.2 Basic AQL Examples

The ASTERIX Query Language (AQL) is a declarative query language similar to XQuery. We list the following simple examples as a gentle introduction. More detailed information on AQL can be found in [17].

6.2.2.1 Scanning a Dataset

```
for $u in dataset('FacebookUsers')
where $u.name = 'Brent Spiner'
return $u
```

The above query iterates over the data instances in the "FacebookUsers" dataset, and returns those instances whose value of the name field equals "Brent Spiner". AQL uses the dot syntax "." for accessing fields.

6.2.2.2 Group By, Order By, and Limit

```
for $u in dataset('FacebookUsers')
for $i in $u.interests
group by $interest := $i with $u
order by count($u)
limit 10
return {"interest": $interest, "frequency": count($u)}
```

Here we are computing the ten most frequent interest ids. We first compute the frequency of each interest id by iterating over all Facebook users and their interests, grouping on the interest id, and counting the number of users with each interest. Finally, we order by the frequency of the interest id and limit the output to the top ten most frequent ones.

6.2.2.3 Joining two Datasets

```
for $u in dataset('FacebookUsers')
for $m in dataset('FacebookMessages')
where $u.id = $m.user-id
return {"user": $u, "msg": $m}
```

Suppose there was another dataset called "FacebookMessages" that had an attribute "user-id" referencing the "id" field in the "FacebookUsers" dataset. We could join the two datasets as shown in the query above. The nested iteration with an equality condition will be translated into an equi-join.

6.2.3 Set-Similarity Queries in AQL

There are two ways to express set-similarity queries in ASTERIX. First, one may use one of the built-in similarity functions directly just like any other function. Second, AQL provides a similarity operator "~=" for convenience. The similarity operator is syntactic sugar,

and internally gets rewritten into the appropriate similarity function. ASTERIX currently supports the Jaccard and edit distance functions, illustrated by the following examples.

6.2.3.1 Edit Distance

Figures 6.3 and 6.4 show examples how to issue string-similarity queries in the AQL DML. The "~=" operator compares its left operand to its right operand to see whether they are similar to each other according to the "simfunction" and "simthreshold" active in its scope (or a system default). In the examples, we use edit distance as the "simfunction" and a "simthreshold" of 2, which is equivalent to the function notation "edit-distance() <= 2" as shown in the (b) portion of the figures. Figure 6.3 shows two equivalent queries asking for all Facebook users whose name is within edit distance 2 of "Brett Spiner". They are examples of a string-similarity selection query on the "name" attribute, because one of the operands of "~=" is a constant (one of the arguments to edit-distance() is a constant). On the other hand, the queries in Figure 6.4 show a string-similarity join. It asks for all pairs of users from the "FacebookUsers" and "MySpaceUsers" datasets. We say the query contains a string-similarity join because the operands of "~=" are both variables (both arguments to edit-distance() are variables).

```
use dataverse SocialData;

set simfunction 'edit-distance';
set simthreshold '2';

for $c in dataset('FacebookUsers')
where $c.name ~= 'Brett Spiner'
return $c
```

(a) ~= Notation

```
use dataverse SocialData;

for $c in dataset('FacebookUsers')
where edit-distance($c.name, 'Brett Spiner') <= 2
return $c
```

(b) Function Notation

Figure 6.3: String-similarity selection query on the "name" field using edit distance.

```
use dataverse SocialData;

set simfunction 'edit-distance';
set simthreshold '2';

for $fu in dataset('FacebookUsers')
for $mu in dataset('MyspaceUsers')
where $fu.name ~= $mu.name
return {'FacebookUser': $fu,
        'MyspaceUser': $mu}
```

(a) ~= Notation

```
use dataverse SocialData;

for $fu in dataset('FacebookUsers')
for $mu in dataset('MyspaceUsers')
where edit-distance($fu.name, $mu.name) <= 2
return {'FacebookUser': $fu,
        'MyspaceUser': $mu}
```

(b) Function Notation

Figure 6.4: String-similarity join query on the "name" fields using edit distance.

6.2.3.2 Jaccard Similarity

Analogous to the examples using edit distance, Figures 6.5 and 6.6 show set-similarity selection and join queries based on Jaccard similarity, using both the operator and function notations. The set-similarity selection query in Figure 6.5 is looking for all Facebook users whose interests are similar to the interest set "[7, 9, 10, 22]". Two interest sets are considered similar if their Jaccard similarity is at least 0.8. Figure 6.6 presents a set-similarity join query to obtain all pairs of users with similar interests from the "FacebookUsers" and "MySpaceUsers" datasets.

```
use dataverse SocialData;

set simfunction 'jaccard';
set simthreshold '0.8f';

for $c in dataset('FacebookUsers')
where $c.interests ~= [7, 9, 10, 22]
return $c
```

(a) ~= Notation

```
use dataverse SocialData;

for $c in dataset('FacebookUsers')
where similarity-jaccard($c.interests,
        [7, 9, 10, 22]) >= 0.8f
return $c
```

(b) Function Notation

Figure 6.5: Set-similarity selection query on the "interests" field using Jaccard.

```
use dataverse SocialData;

set simfunction 'jaccard';
set simthreshold '0.8f';

for $fu in dataset('FacebookUsers')
for $mu in dataset('MyspaceUsers')
where $fu.interests ~= $mu.interests
return {'FacebookUser': $fu,
        'MyspaceUser': $mu}
```

(a) ~= Notation

```
use dataverse SocialData;

for $fu in dataset('FacebookUsers')
for $mu in dataset('MyspaceUsers')
where similarity-jaccard($fu.interests,
        $mu.interests) >= 0.8f
return {'FacebookUser': $fu,
        'MyspaceUser': $mu}
```

(b) Function Notation

Figure 6.6: Set-similarity join query on the "interests" field using Jaccard.

6.2.4 Secondary Indexes for Set-Similarity Queries

ASTERIX supports the creation of secondary indexes via its DDL. There are four variants of inverted indexes that support answering set-similarity queries, listed below with DDL examples:

1. **Keyword Index:** An inverted index mapping from word tokens or list elements to the corresponding primary key of that data item. They can be created on string or list field types (ordered and unordered lists are supported). String types are tokenized into words. The following is a DDL example:

   ```
   create index interests_ix on FacebookUsers(interests) type keyword;
   ```

2. **Fuzzy Keyword Index:** A keyword index optimized for answering set-similarity queries using length partitioning (as described in Section 5.3.2). The following is a DDL example:

   ```
   create index interests_ix on FacebookUsers(interests) type fuzzy keyword;
   ```

3. **N-Gram Index:** An inverted index mapping from the n-gram tokens (synonymous to q-grams) of a string field to the primary key of the corresponding data item. The

following example shows DDL to create a 3-gram inverted index on the "name" field of our FacebookUsers dataset:

```
create index name_ix on FacebookUsers(name) type ngram(3);
```

4. **Fuzzy N-Gram Index:** A length-partitioned n-gram index optimized for set-similarity queries. The following is a DDL example:

```
create index name_ix on FacebookUsers(name) type fuzzy ngram(3);
```

The main difference between the length-partitioned ("fuzzy") and regular versions of those indexes is that the length-partitioned versions are more performant for queries that benefit from length filtering (Section 5.3.2), whereas regular versions are more performant for queries that cannot benefit from length filtering, e.g., conjunctive keyword queries. While the fuzzy indexes do support answering such queries, doing so may incur a significant performance hit because all length partitions must be considered during an index search. Similarly, the regular index versions also support set-similarity queries, but are typically not as efficient as the "fuzzy" versions. Due to this performance tradeoff, ASTERIX supports both fuzzy and non-fuzzy inverted indexes.

6.2.4.1 Index Compatibility Matrix

The following table lists which similarity functions can be optimized with which types of indexes. The compatibility matrix below applies to the "fuzzy" and regular index versions, and also to both selection and join queries.

For example, the Jaccard-based queries in Figures 6.5 and 6.6 could be optimized by a keyword index on the "interests" field. On the other hand, the string-similarity queries based on edit distance in Figures 6.3 and 6.4 could not be optimized by a keyword index, but require an N-Gram index type.

Index Type (Type of Indexed Field)	Edit Distance	Jaccard
Keyword (String)	N	Y*
Keyword (Ordered List)	N	Y
Keyword (Unordered List)	Y	Y
N-Gram (String)	Y	Y*

Table 6.1: Secondary-index compatibility matrix for similarity functions. *Requires matching word or n-gram tokenizer on indexed field.

6.3 Answering Set-Similarity Queries Using Indexes

In this section, we show how to rewrite and execute set-similarity selection and join queries in ASTERIX using secondary indexes. We present a few plan alternatives and discuss their tradeoffs in preparation for our experimental evaluation.

After undergoing parsing and analysis, an AQL query is rewritten into a logical operator tree. At this stage, the rule-based optimizer (Algebricks) performs a pre-order traversal of the logical operator tree and attempts to apply a set of rewrite rules at each operator. A rewrite rule matches a pattern in the logical operator tree, and if applicable, adds, removes, and/or replaces operators with another set of operators. This process repeats until a fixed point is reached (with heuristic guidance to guarantee termination). The following illustrations are simplified for clarity.

6.3.1 Rewriting Selection Queries

Figure 6.7 shows how a set-similarity selection query is optimized with an index, assuming there is an applicable one.

The left-hand side shows the original query plan, and the right-hand side shows the optimized plan. Based on a selection operator with a similarity condition (using the "$\sim=$" notation), we wish to replace the primary-index scan (PIX scan) with a secondary-index based plan. The

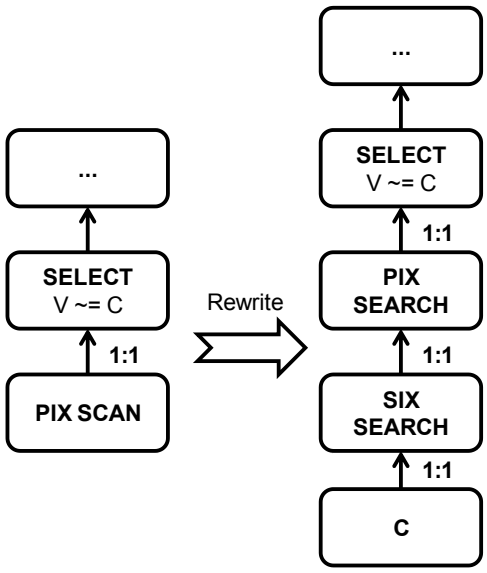


Figure 6.7: Selection-query plan rewritten with an index. "PIX" stands for Primary Index, and "SIX" for Secondary Index. The select operator contains a similarity condition on a field value "V" and a constant value "C". The 1:1 annotations indicate that there is no data redistribution across data partitions. The query is sent to all secondary-index partitions, and each partition executes its local operator pipeline in parallel.

optimized plan first feeds the constant value "C" into the secondary index to obtain a set of qualifying primary keys, and then looks up those primary keys in the primary index to fetch their corresponding data items. Notice that we do not remove the select operator from the optimized plan. The reason is that for set-similarity queries, our secondary indexes produce a set of primary keys satisfying the T-occurrence condition (Section 2.4), but not necessarily the original similarity condition. Therefore, we still need to compute the final similarity after searching the primary index to remove false positives. If the similarity condition is selective enough, such an index-based plan is often much more efficient than a scan-based one. Not only can we avoid a costly data scan, but we can also reduce the number of data items we must evaluate the (possibly expensive) similarity function against.

The basic flow of the rewrite rule for optimizing set-similarity selection queries is depicted in Figure 6.8. We must first match an operator pattern that consists of a linear pipeline with a select operator and a primary-index scan operator. Next, we analyze the select

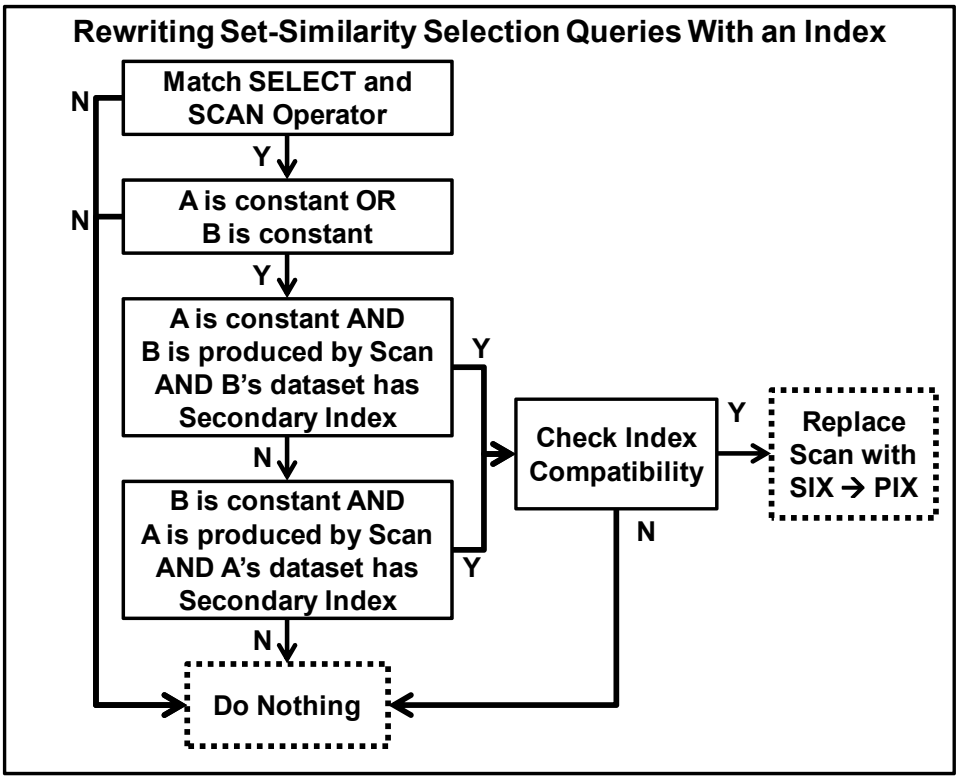


Figure 6.8: Rewrite rule for optimizing a selection query. "PIX" stands for Primary Index, and "SIX" for Secondary Index.

operator's condition to see if it contains a similarity condition, and if one of its arguments is a constant. If so, we continue the analysis to determine whether the non-constant argument originates from the primary-index scan operator, and whether the corresponding dataset has secondary indexes on "B". For each secondary index on "B", we check the compatibility matrix (Table 6.1) to determine its final applicability.

6.3.1.1 Panic Cases with Edit Distance

Recall that for queries using edit-distance, the lower bound on the number of common q-grams (or tokens) may become zero or negative (Section 2.4.1). For such "panic" cases we must revert to a scan-based plan, even if an otherwise applicable index is available. For selection queries, we can determine such panic cases at query compile time when applying

the corresponding index rewrite rule by analyzing the constant argument to the similarity condition. If we detect a panic case, then we simply bail from the rewrite rule. On the other hand, we will see that for set-similarity joins such panic cases must be dealt with at run-time, unfortunately (see Section 6.3.3.1). Note that no such panic cases are possible for similarity queries based on Jaccard, because if two sets have no elements in common, then they can never reach a Jaccard similarity greater than 0. In contrast, two strings could be within a certain edit distance even if the strings' q-gram sets have no common elements.

6.3.2 Parallel Execution of Index-Based Selection Queries

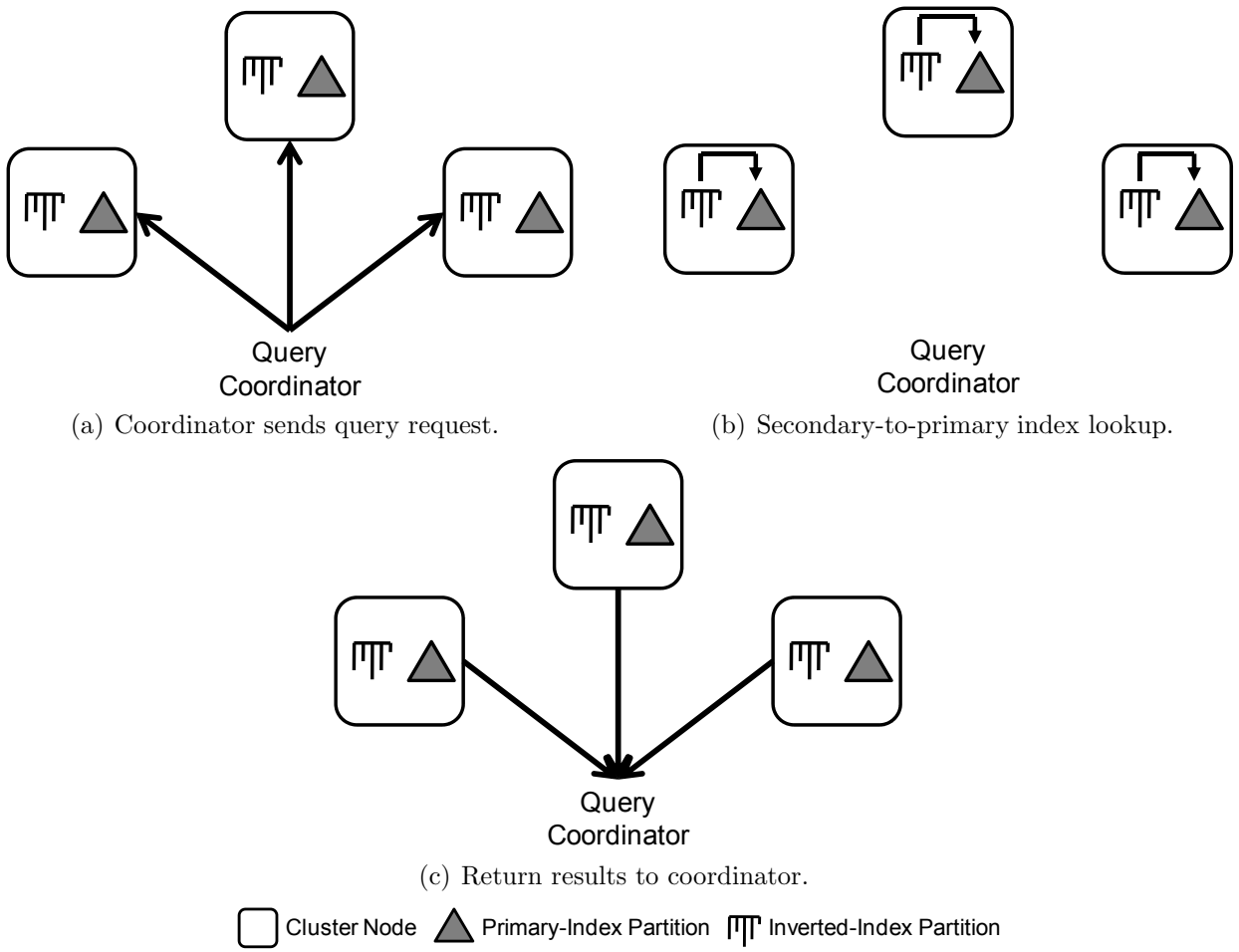
The optimized plan in Figure 6.7 using a secondary index will be executed as follows. Recall that ASTERIX currently supports local secondary indexes (see Section 6.1.2) whose partitions are always co-located with their corresponding primary-index partitions.

Figure 6.9 shows a cluster with 3 nodes (excluding the coordinator node) that house a partitioned primary index and a local inverted index. To answer a set-similarity selection query, the coordinator sends requests to the nodes of all partitions (Figure 6.9(a)). The query request itself contains the constant search key to be fed into the secondary index. At each node, we probe the inverted index for a set of primary keys and lookup those primary keys in the co-located primary index (Figure 6.9(b)). Finally, each participating node sends back its results to the coordinator (Figure 6.9(c)).

6.3.3 Rewriting Join Queries

The basic rewriting of set-similarity join queries using indexes is shown in Figure 6.7.

We say the optimized plan on the right-hand side uses an "indexed nested loops" join strategy. Similar to the rewrite for selection queries, we replace the primary-index scan of one



(a) Coordinator sends query request.

(b) Secondary-to-primary index lookup.

(c) Return results to coordinator.

Figure 6.9: Parallel execution of a selection query using a local secondary index.

side of the join with a secondary-index search followed by a primary-index search. It is only required that one side of the join is sourced at a primary-index scan. The other side could be an arbitrary operator subtree ("SUB TREE" in the figure). We call the side of the join that is rewritten with a secondary index the "index side", and the other side the "probe side". In the optimized plan, the probe side feeds into the secondary-index search operator. That is, every tuple from "SUB TREE" will act as a search key to the secondary index (or more precisely some of the tuples' fields, as appropriate). As before, we need to remove false-positive answers from the index-based subplan with a select operator on the original similarity condition, which is taken from the join operator. Notice the "M:N" connection between the probe subtree and the secondary-index search. It signifies that each partition

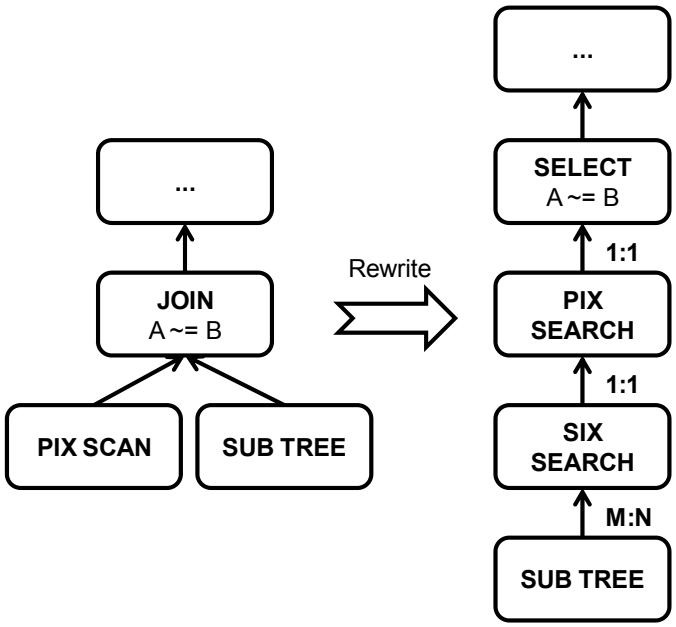


Figure 6.10: Join-query plan rewritten with an index. "PIX" stands for Primary Index, and "SIX" for Secondary Index. The select operator contains a similarity condition, and we use the similarity-operator notation for simplicity. The "1:1" annotations indicate that there is no data redistribution across data partitions. The "M:N" annotation indicates a broadcast data exchange, i.e., each partition sends its data to all other partitions.

executing the "SUB TREE" plan must broadcast its output tuples to all secondary-index partitions. We will shortly discuss the execution of the index-based query plan in more detail in Section 6.3.4.

Figure 6.11 outlines the rewrite rule for introducing indexed nested loops joins. We first match the required operator pattern that consists of a join that has at least one of its inputs sourced at a primary-index scan. Next, we analyze the join condition to make sure the similarity function has two non-constant arguments. If so, we continue by checking if either the A or B argument of the similarity condition is produced by the join input sourced at the primary-index scan, and whether the corresponding dataset has applicable secondary indexes. Finally, we consult the index compatiblity matrix to decide whether rewriting the query using an index is possible or not.

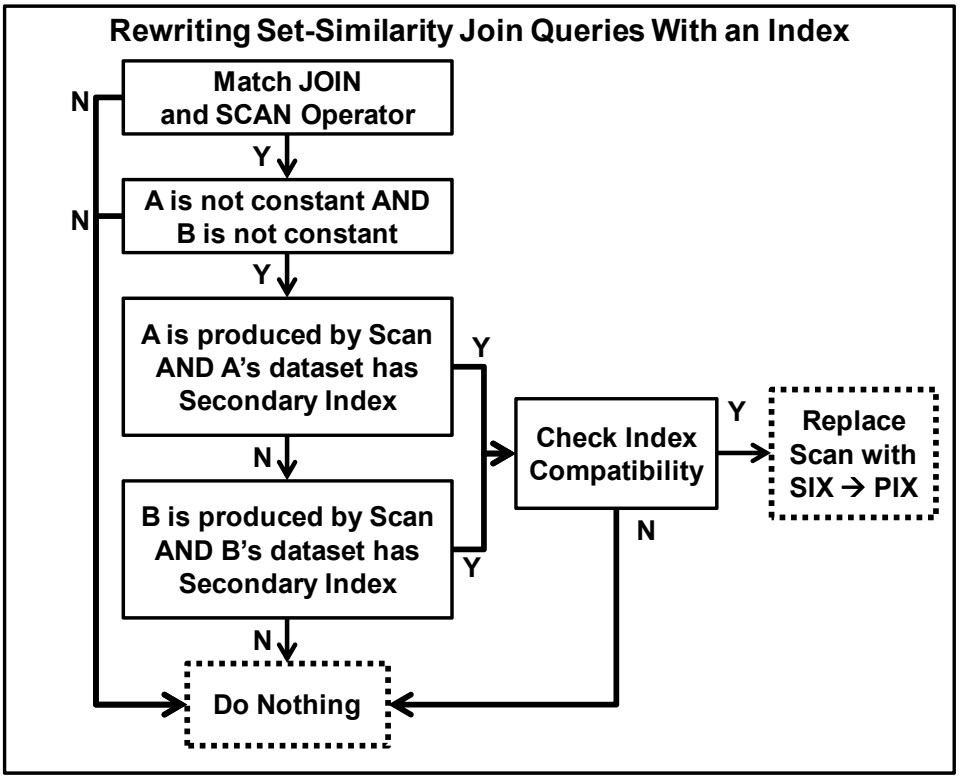


Figure 6.11: Rewrite rule for optimizing a join query. "PIX" stands for Primary Index, and "SIX" for Secondary Index.

6.3.3.1 Panic Cases with Edit Distance

For string-similarity queries using edit distance, we must modify the basic indexed nested-loops join plan from Figure 6.11 to correctly handle "panic cases" (Section 2.4.1). Unlike selection queries where the secondary-index search key is always a constant, the secondary-index search keys for a nested-loops join are produced by the probe input branch ("SUB TREE"). Therefore, panic cases need to be dealt with at query runtime, as opposed to query compile time for selection queries.

Figure 6.12 shows the modified indexed nestes-loops join plan for correctly handling panic cases when using edit distance. The main difference lies in separating the tuples produced by the probe subtree into two sets, one that contains non-panic tuples ($T > 0$), and one that contains panic tuples ($T \leq 0$). We perform this separation via a split operator above the

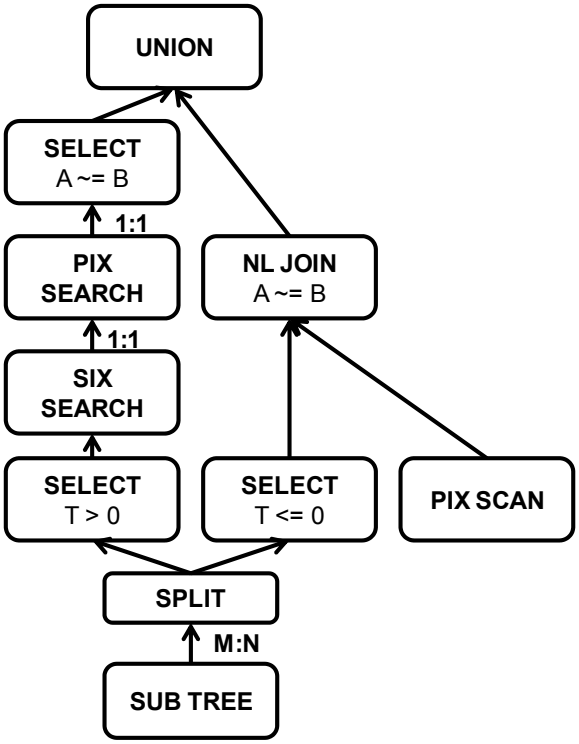


Figure 6.12: Indexed nested-loops join plan for handling panic cases with edit distance. The probe input is split into two sets, one that contains panic tuples, and one that contains non-panic tuples. "T" stands for the T-occurrence threshold.

probe subtree, followed by a selection operator on each of the its two output branches, which filter out the panic and non-panic tuples, respectively. The non-panic tuples are fed into the secondary-to-primary index plan as before. The panic tuples participate in a (non-indexed) nested-loops join plan. The final query result is the union of the results of those two joins.

6.3.4 Parallel Execution of Index-Based Join Queries

Suppose we wanted to run a simple set-similarity join such as the one from Figure 6.6 that joins a FacebookUsers dataset with a MySpaceUsers dataset based on the Jaccard similarity of their interest fields. Further, suppose that there is a secondary index supporting Jaccard on the FacebookUsers dataset. The execution of an indexed nested-loops join based on such a local secondary index is illustrated in Figure 6.13 using a 3-node cluster. First,

the coordinator sends the query request to all relevant nodes, namely those that house partitions of the FacebookUsers and/or MySpaceUsers datasets (Figure 6.13(a)). The query-operator pipeline starts by initiating a parallel scan on the probe side (MySpaceUsers) and broadcasting its tuples to all other nodes to ultimately search each secondary-index partition with *all* tuples from the probe side (Figure 6.13(b)). The broadcast is necessary because the secondary index is co-partitioned with its primary index (FacebookUsers), and no partition pruning is possible based on the secondary-index key (the "interests" field). Figure 6.13(c) shows the pipelined execution of the secondary-to-primary index search plan; it is fed from local as well as remote probe-side partitions. Finally, once each secondary-index partition has processed all tuples from the probe side, we return the results to the coodinator. (The writing of the results could also be pipelined with the primary-to-secondary index plan.)

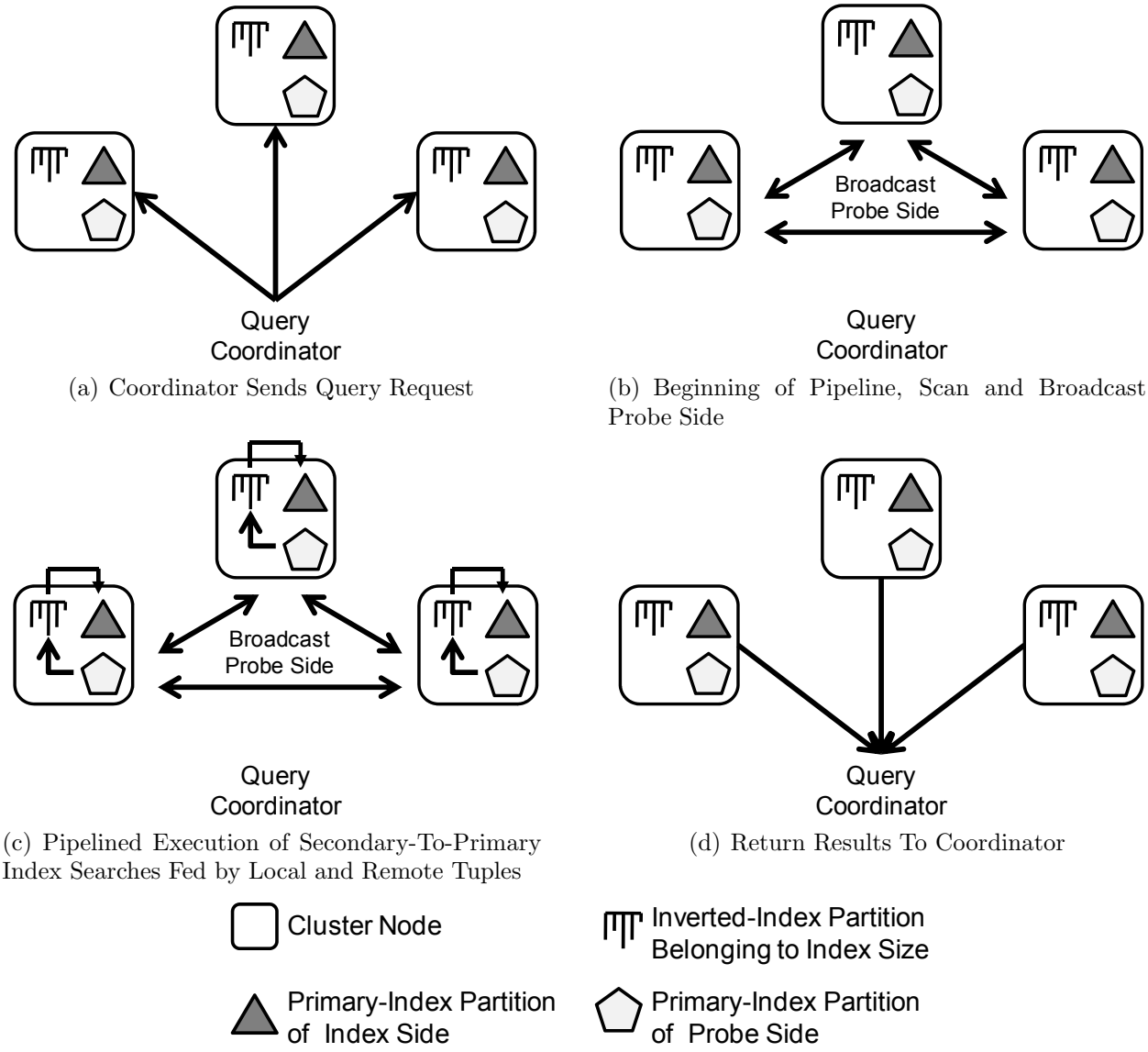
6.3.5 Plan Optimizations/Alternatives

Next, we present two possible query-plan improvements whose merit will be evaluated experimentally in Section 6.5.

6.3.5.1 Sorting The Primary Keys

One simple optimization for secondary-index based plans is to sort the primary keys resulting from the secondary-index search before looking them up in the primary index. The main goal of this optimization is to improve buffercache hit rates during the series of primary-index searches. The buffercache is responsible for transferring data pages to/from disk and for caching such pages in memory. Figure 6.14 shows the index-based plan of (a) a selection and (b) a join query including a sort before the primary-index search.

It is not obvious whether employing such a sort is generally beneficial or not for set-similarity



(a) Coordinator Sends Query Request

(b) Beginning of Pipeline, Scan and Broadcast Probe Side

(c) Pipelined Execution of Secondary-To-Primary Index Searches Fed by Local and Remote Tuples

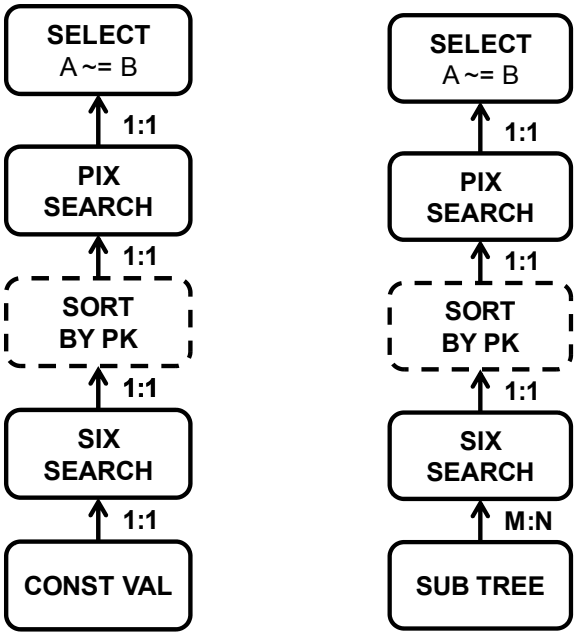
(d) Return Results To Coordinator

Figure 6.13: Parallel execution of a join query using a local secondary index.

queries. Based on the following advantages and disadvantages, we believe the benefit of sorting is very much query and data dependent. We therefore investigate this question experimentally in Section 6.5.

Advantages of sorting:

- Improve buffercache hit rate during primary-index searches.



(a) Index-Based Selection Query

(b) Index-Based Join Query

Figure 6.14: Index-based query plans optimized by sorting on primary keys. The optional sort operator is show with a dashed lines.

- May enable sort-based operators in the downstream plan.

Disadvantages of sorting:

- Sorting requires extra work possibly involving I/O.
- Prevents pipelining the secondary and primary index searches, because the sort must complete before primary-index searches can commence.
- Requires extra memory for the sort operator.
- Sorting may not significantly improve the buffercache hit rate if the number of primary keys to search is small.

6.3.5.2 Surrogate-Based Indexed Nested-Loops Join

The major drawback of an indexed nested-loops join using a local secondary index is the need to broadcast the probe side to all secondary-index partitions (as explained in Section 6.3.4). This broadcast operation is a direct consequence of the co-partitioning of the secondary with its primary index, and therefore, it cannot be eliminated without changing the partitioning strategy of the secondary index. However, we may reduce the cost of the broadcast by only transferring the minimal information required to search the secondary index. The main idea is to only send the secondary-key fields together with a compact *surrogate* for each probe-side tuple, such that we can later resolve the surrogates to obtain their original, complete tuples. This idea is reminiscent of semi-join optimizations in distributed databases [67]. For example, a primary key is a common surrogate, but if none is available, one could also generate an artificial surrogate id on-the-fly.

Figure 6.15 shows a surrogate-based indexed nested-loops join plan. Notice the project operator after the probe subtree, which eliminates all non-essential fields from the probe side's tuples. Not only does such a surrogate-based plan reduce network transfer costs (of the broadcast), but it also reduces data copying in the secondary-to-primary index operator pipeline, during which the result cardinality is likely to increase (as for any join). After the secondary-to-primary index searches we must resolve the surrogates from the probe side to obtain their complete tuples. As show in the figure, we resolve the surrogates with a top-level join of the original probe subtree with the indexed nested-loops subtree (after removing false positives). We could either execute the original probe subtree twice, e.g., if it is something simple such as a scan, or we could materialize the probe subtree's results to subsequently feed them into the different branches in the plan. Since the top-level join is an equi-join on the surrogates, S_L and S_R, it can be executed efficiently in parallel, e.g., using a hash join.

The following is a list of advantages and disadvantages of employing a surrogate-based join

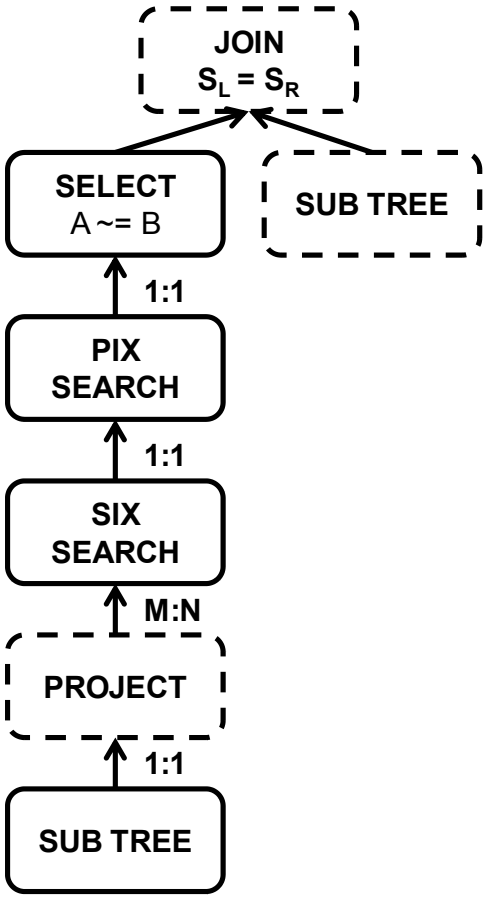


Figure 6.15: Surrogate-based indexed nested-loops join plan. The new operators are shown with dashed lines. The bottom project operator removes all fields except for those used as keys in the secondary-index search, as well as a unique identifier (a *surrogate*) for tuples from "SUB TREE". The top-level join retrieves the complete records from "SUB TREE" by equi-joining on the surrogates from the left side (S_L) and the those from the right (S_R).

strategy using a secondary index:

Advantages of using surrogates:

- Reduce data transferred over network during broadcast of the probe side.
- Reduce bytes copied in the secondary-to-primary operator pipeline.

Disadvantages of using surrogates:

- Requires additional, possibly pipeline-breaking, join to resolve surrogates.
- Only mitigates fundamental problem of broadcast, but does not solve it.
- Requires efficient way to generate surrogates if no primary key(s) are available.

Understanding the performance tradeoffs between the regular and surrogate-based plans is one of the goals of our experimental evaluation in Section 6.5.

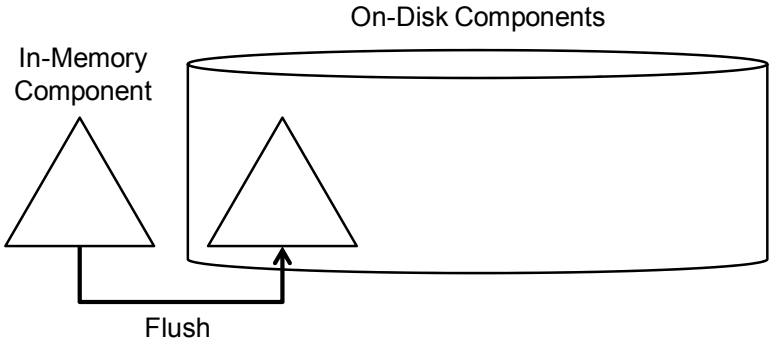
6.4 Inverted Indexes in ASTERIX

In this section we discuss the implementation of the inverted indexes used in ASTERIX to answer set-similarity queries. We introduce a basic inverted index with its components and supported operations, and show how its improved length-partitioned version (see Section 6.2.4) incorporates length filtering. Finally, we present the query plans for constructing such indexes.

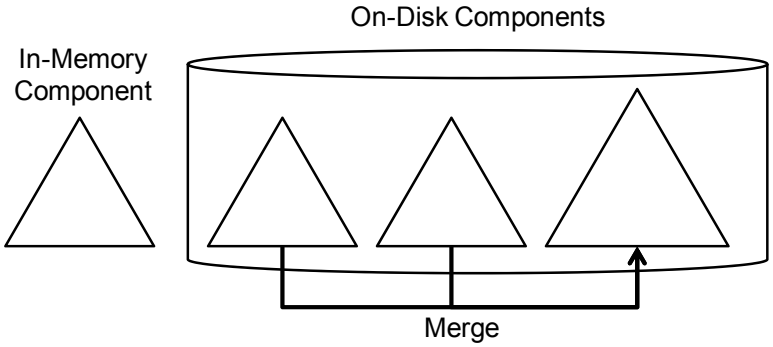
6.4.1 Log Structured Merge Framework

The storage subsystem in ASTERIX is based on ideas from the Log Structured Merge (LSM) Tree [100]. The LSM-Tree is a totally ordered, persistent index that supports typical operations such as insert, delete, and search. Its main goal is to optimize write operations at the expense of read operations, e.g., as compared to a B-Tree where each insert/delete may incur 2 disk I/Os (one for reading a page, one for writing). To that end, updates to an LSM-Tree are staged into an in-memory buffer whose contents are periodically flushed to disk in large chunks to amortize the disk usage over multiple updates. Performing index maintenance in such a "batched" fashion is common practice for inverted indexes specifically [74, 126], due to the multiplicity of updates (adding one document changes many inverted lists). Several

strategies on how to exactly transfer in-memory data to disk, and how to organize the disk-resident data have been proposed [100, 58, 105], most of them involving a single mutable in-memory index component and multiple immutable disk-resident index components. As a consequence, the search performance of such a multi-component index may be worse than comparable single-component indexes such as a standard B-Tree.



(a) Flushing a full in-memory component to disk.



(b) Merging multiple on-disk components into a single component.

Figure 6.16: Basic LSM Operations: Flush and Merge.

We will describe the embodiment of these ideas in ASTERIX focusing on the implementation of our inverted index. The basic framework and component-level operations for an LSM-based index in ASTERIX are shown in Figure 6.16. Each LSM-index has a mutable in-memory component that is backed by a fixed amount of memory. Once the in-memory component has exhausted its space due to insert and/or delete operations, we write the entire component to disk, creating a new on-disk component. Such a "flush" operation as shown in Figure 6.16(a) frees up the in-memory component to accommodate new updates.

Since index-search operations must consider all components, including the one in memory and those on disk, the performance of such searches degrades with every new component. We mitigate this issue by periodically merging several on-disk components into a single, larger component as shown in Figure 6.16(b). We remove the old components after a merge has completed successfully. ASTERIX currently follows a simple strategy of scheduling a merge operation once there are three or more on-disk components, and we leave a more detailed discussion of flush and merge policies to future work. Next, we discuss our inverted index implementation in the context of this LSM framework.

6.4.2 Inverted Index Implementation

6.4.2.1 Inverted Index Components

Figure 6.17 shows a three-component inverted index with one mutable in-memory component and two immutable on-disk components.

The in-memory component consists of two in-memory B-Trees that share the allocated memory budget. We flush the in-memory component to disk once its two B-Trees have jointly exhausted the memory budget. The "insert" B-Tree shown in light gray stores and is ordered by `(token, primary key)` pairs, each one indicating that the indexed data item identified by a primary key contains the associated token. That B-Tree acts as an in-memory inverted index, and one may iterate over the "inverted list" of a particular token using a prefix search on the B-Tree. Similarity, one may perform a random-access search for a specific item in an "inverted list" by probing the B-Tree with a composite `(token, primary key)` key. To further clarify, Figure 6.18 shows an excerpt of an in-memory inverted index on the 2-gram tokens of the string collection shown on the left-hand side ("id" is the primary key).

We decided to implement the in-memory inverted index using a B-Tree mainly to avoid the

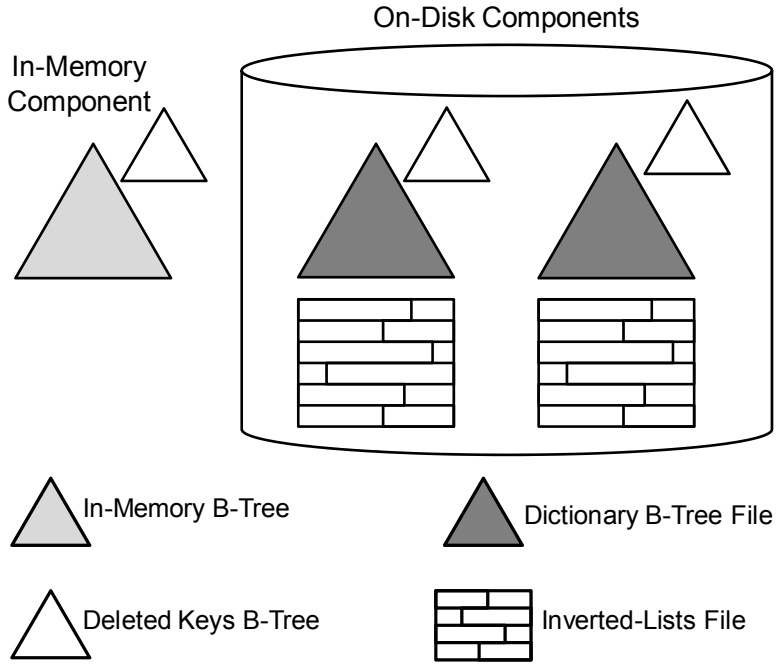


Figure 6.17: Overview of a three-component inverted index.

engineering complexity of creating a special-purpose in-memory inverted index. In particular, our existing B-Tree naturally provides the following desiderata for an in-memory inverted index:

- Efficient mutlithreading using a variant of the Aries/IM [95] latch protocol.
- Ordered retrieval. This feature is useful for the flush operation, since bulk-loading an on-disk inverted index relies on a total order.

Data Strings

id	String
1	cat
2	cathey
3	kathy
4	kat
5	cathy

In-Memory Inverted Index

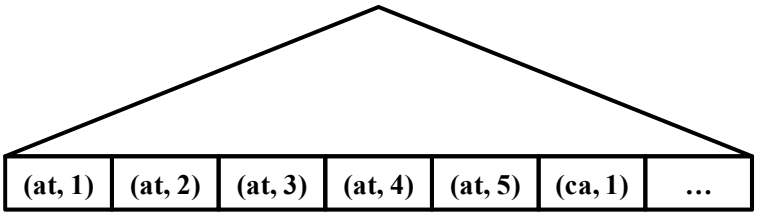


Figure 6.18: Excerpt of an in-memory inverted-index component on 2-gram tokens.

- Since a B-Tree is paginated, it is natural to impose a memory bound.

The "deleted keys" B-Tree contains primary keys that have been deleted and thus should not be returned by index searches. The primary keys in the in-memory deleted-keys B-Tree refer to primary keys of on-disk components, and *not* to keys in the in-memory "insert" B-Tree. We remove documents from the in-memory "insert" B-Tree by directly deleting the corresponding `(token, primary key)` pairs. As a result, every deleted-keys B-Tree refers to primary keys contained in components that are older than itself (i.e., flushed at an earlier time).

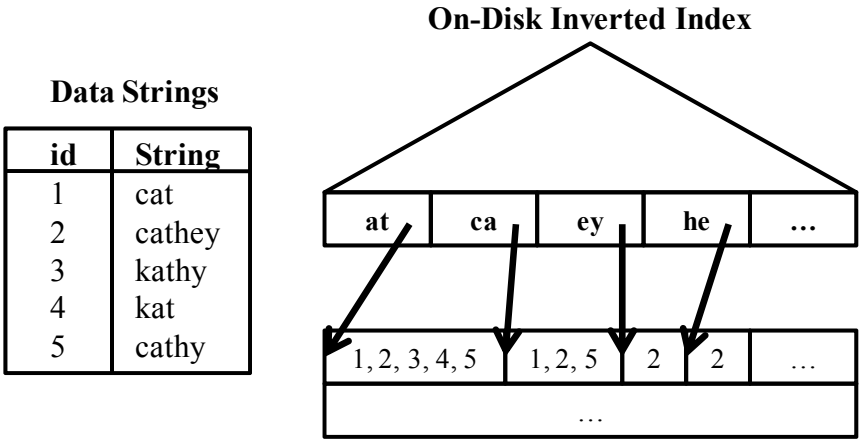


Figure 6.19: Excerpt of an on-disk inverted-index component on 2-gram tokens.

In addition to a persistent deleted-keys B-Tree, the on-disk components of our inverted index use a dictionary B-Tree and a separate file containing inverted lists. The dictionary B-Tree contains `(token, inverted-list pointer)` pairs, as shown in Figure 6.19. An inverted-list pointer directs to the beginning of an inverted list in the corresponding inverted-lists file, and also records the length of the list. Each inverted list contains a list of ordered primary keys and is stored contiguously in the file. While it is conceivable that the tokens and inverted lists could be stored efficiently in a single B-Tree using RID-lists [43], we chose to separate the tokens from the inverted lists mainly for simplicity (separation of concerns), as well as the following reasons:

- We hope to be able to cache most of the token dictionary in memory, improving buffercache hit rates, since it is often unreasonable to assume a large portion of the inverted lists can fit into memory.
- To enable optimizations that avoid reading inverted lists entirely, such as those from Section 5.4.
- To efficiently answer queries that ask for token frequencies using only the dictionary.

6.4.2.2 Inverted Index Operations

The following listing explains the four basic inverted-index operations. We refer to a "document" as some data field that can be tokenized (or is already a list) and used as a key in an ordered index.

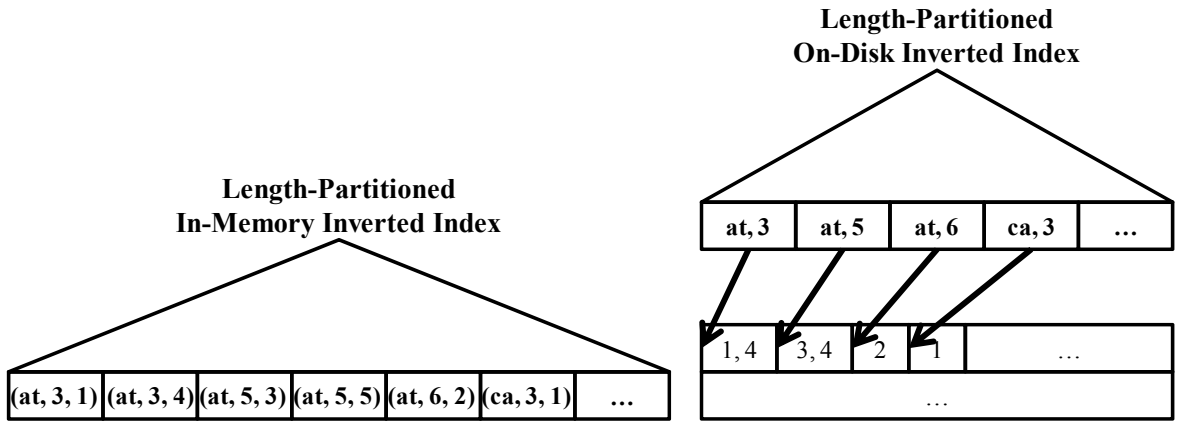
- **Bulk-Load:** Given a sorted stream of `(token, primary key)` pairs, the bulk load operation creates a single on-disk inverted-index component. The bulk load operation consumes the pairs one-by-one, appending the primary keys to the inverted-lists file. Whenever it completes writing one inverted list, by detecting a change in the token, the procedure generates a `(token, inverted-list pointer)` for the dictionary B-Tree. Such pairs are appended to the dictionary B-Tree using a standard bottom-up bulk-loading method.
- **Insert:** Given a document and its primary key, the insert operation tokenizes the document according to the type of inverted index (see Section 6.2.4), and inserts `(token, primary key)` pairs into the in-memory component (using standard B-Tree inserts).
- **Delete:** Given a document and its primary key, the delete operation tokenizes the document, deletes all resulting `(token, primary key)` pairs from the in-memory com-

ponent, and inserts the primary key into the in-memory deleted-keys B-Tree, since the key may be present in older components.

- **Search:** The search operation expects an appropriate search predicate, such as a document, a similarity function, and a similarity threshold. The procedure processes each index component one-by-one. For each component we solve the T-occurrence problem (see Section 2.4.1) to obtain a set of candidate results. For each such candidate, we probe the deleted-keys B-Trees of *all* newer components to remove deleted documents from the set of results. Recall that the set of results still contains false-positive answers which need to be removed by a select operator at the query-plan level (see Section 6.3).

6.4.2.3 Length-Partitioned Inverted Index Implementation

Based on our previous results from Section 5, we have also added support for length-partitioned inverted indexes in ASTERIX, called "fuzzy" inverted indexes (Section 6.2.4). Figures 6.20(a) shows a portion of a length-partitioned in-memory inverted index, and Figure 6.20(b) its on-disk counterpart. They correspond to the unpartitioned versions in Figures 6.18 and 6.19, respectively, and are likewise constructed on the 2-grams of our example string collection. The partitioned in-memory index now contains triples of `(token, length, primary key)` to perform length filtering. For example, the triple `(at, 5, 3)` means that the data item with primary key 3 contains the 2-gram `at` and has string-length 5. To perform index searches, the portion of an inverted list within a certain length range can be obtained by doing a prefix search on the B-Tree. The representation of an on-disk length-partitioned component is very similar to the one presented in Section 5.3. The main difference is that instead of the FilterTree structure, we use the dictionary B-Tree to map from a `(token, length)` pair to an inverted list in the inverted-lists file. For example, in Figure 6.20(b) the dictionary B-Tree stores a pair `(at, 5)`, which points to an inverted list containing the primary keys of strings of length 5 that contain the 2-gram `at`.



(a) In memory. (b) On disk.

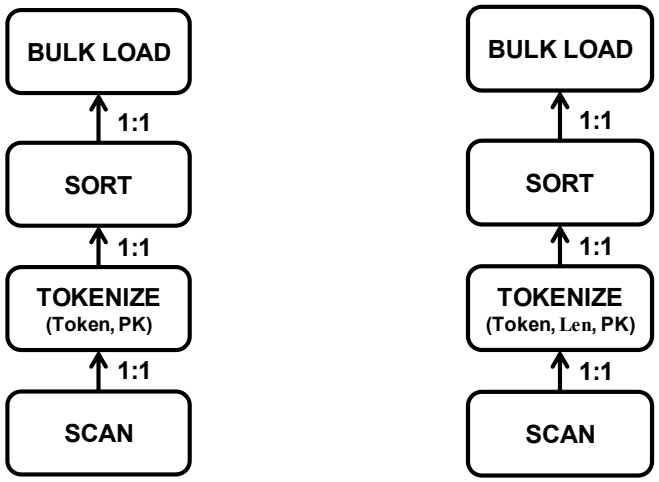
Figure 6.20: Length-partitioned index components.

Differences from solutions in Chapter 5: The implementation of the length-partitioned inverted index in ASTERIX differs from the solutions discussed in Chapter 5 in two ways. First, because we expect insert-heavy workloads in ASTERIX it may be detrimental to the overall system performance to create a separate index containing a projection of the indexed field, called "dense index" in Chapter 5. Such a dense index would possibly improve read-query performance, however, at the price of more costly inserts, as we must maintain the additional dense index as well. Second, we did not implement the cost-based adaptive algorithm for searching a length-partitioned inverted index in ASTERIX because, as shown in Chapter 5, there is only a marginal benefit of using that search algorithm without a proper dense index.

6.4.3 Inverted Index Construction

Creating a secondary index on an ASTERIX dataset via a DDL statement (Section 6.2.4) generates a job to bulk-load the new index based on the existing data in the dataset. Recall that bulk-loading an inverted index will create a single, disk-resident component. The query plan to perform this initial loading is based on external sorting, as shown in Figure 6.21.

We scan each partition of the source dataset in parallel, tokenize the indexed field, and then sort the resulting pairs (or triples) to finally feed an inverted-index bulk-load operator. The only difference between creating a regular versus a fuzzy inverted index is that the tokenizer generates `(token, primary key)` pairs for a regular index, and `(token, length, primary key)` triples for a fuzzy one. As a result, constructing fuzzy inverted indexes is more expensive because of the additional amount of data (the length) that also needs to be sorted. Since ASTERIX uses local indexes, the index-construction procedure is performed locally (in parallel) on each data partition as indicated by the 1:1 connections in the figure.



(a) Regular inverted index. (b) Length-partitioned inverted index.

Figure 6.21: Query plans for constructing inverted indexes.

6.5 Experiments

6.5.1 Cluster Configuration

We conducted our experiments on a 10-node IBM cluster whose configuration is summarized in Table 6.2. Each node had a quad-core Intel Xeon E5520 CPU at 2.26GHz, 12GB of RAM, and four 300GB hard drives. In total, the cluster had 40 disks and 40 CPU cores and an

aggregate of 120GB of RAM.

System Configuration	
Number of Nodes	10
Total Number of Disks	40
Total Number of CPU Cores	40
Total System RAM	120GB
Node Configuration	
CPU	4-Core Intel Xeon E5520, 2.26Ghz
RAM	12GB
Disk	4x300GB, 10k RPM, SATA drives
Network	1Gbit Ethernet
Operating System	CentOS 5.5
JVM	OpenJDK 64-Bit Server

Table 6.2: IBM Cluster Configuration.

6.5.2 ASTERIX Configuration

ASTERIX is a layered system with Algebricks and Hyracks underneath. Different components of the system allocate memory from the JVM heap in different ways. The BufferCache is responsible for transferring disk pages to/from disk, and it has a fixed number of buffers for caching disk pages. There is one BufferCache per Hyracks Node Controller, and its memory buffers are allocated once from the JVM heap when ASTERIX starts up. On the other hand, Hyracks operators allocate their working memory in chunks (Hyracks frames) directly from the JVM as needed, and once those operators complete, their memory is available to be freed by the garbage collector. As a consequence of this design, the BufferCache cannot dynamically expand/contract and Hyracks operators cannot use possibly unused memory from the BufferCache. Therefore, we carefully carved up RAM in our experiments as shown in Table 6.3. Of the 12GB total memory on each Node Controller we left 2GB for the operating system and gave the remaining 10GB to the JVM running the Node Controller. Of those 10GB, we left 2GB available for Java objects, classes, etc. We split the remaining 8GB evenly among the BufferCache and the memory used for running Hyracks operators.

Total System Memory	12GB
Memory for OS	2GB
JVM Heap Size	10GB
Java Objects, etc.	2GB
Buffer Cache Size	4GB
Memory for Operators	4GB
Sort Buffer Size	256MB
GroupBy Buffer Size	256MB
Join Buffer Size	256MB
Hyracks Frame Size	128KB
Buffer Cache Page Size	128KB

Table 6.3: ASTERIX Memory Configuration.

For determining how much memory to allocate to each Hyracks operator, we examined the query plan of the unindexed set-similarity join [110], which we used for comparison in our experiments. The query plan for that join method runs up to four operators that require memory concurrently (one sort, two joins, and one groupby). Each Node Controller stored four partitions of the data, meaning that during the execution of such an ad-hoc set-similarity join, there may be 4*4=16 concurrent operators that require memory. We decided to split the 4GB for Hyracks operators evenly among those 16 operators, resulting in 4GB / 16 = 256MB for each operator. We used these settings for all of the following experiments.

6.5.2.1 Data Loading and Index Creation

For each experiment, we loaded the data into native ASTERIX datasets using AQL's bulk-loading utility. In the context of our LSM framework (See Section 6.4.1), bulk-loading a dataset results in one LSM-based primary B-Tree with a single on-disk component per partition. Similarly, our secondary inverted indexes consisted of a single persistent LSM component after being loaded.

6.5.2.2 Query Submission

Each node ran one Hyracks Node Controller, and one of the nodes ran the Hyracks Cluster Controller. We configured ASTERIX to start up a Web server that accepted queries via HTTP requests. The Web server ran on the same machine as the Hyracks Cluster Controller. The servlet to which ASTERIX queries were issued performed ASTERIX query compilation and then submitted the resulting Hyracks job to the Cluster Controller for execution. We submitted AQL queries to ASTERIX via an external client machine that was not part of the IBM cluster. All reported response times reflect the end-to-end time, starting from the client's query request and ending when the client received a response of query completion.

6.5.3 Datasets

We conducted our experiments on the following real-world and synthetic datasets.

6.5.3.1 Real Data: DBLP and Citeseer

DBLP[1]**:** This dataset contained information on 1.2 million publications, including titles, authors, publication journal or conference, etc. We followed the same preparatory steps as those in [109] to transform the original XML file into a CSV file intended to be loaded into an ASTERIX dataset with the schema shown in Figure 6.22. The DBLP dataset consisted of an artificial unique "id" field used as its primary key and hash-partitioning key, a "dblpid" used as a descriptive identifier in DBLP, the publication title, and a string field "authors" containing all of a publication's authors separated by commas. Finally, the "misc" field was formed by concatenation of the remaining fields from the original XML file (publication medium, publication date, and journal or conference).

[1] http://dblp.uni-trier.de/xml/dblp.xml.gz

```
create type DBLPType as closed {
  id: int32,
  dblpid: string,
  title: string,
  authors: string,
  misc: string
}
```

```
create type CSXType as closed {
  id: int32,
  csxid: string,
  title: string,
  authors: string,
  misc: string
}
```

Figure 6.22: ASTERIX schema definition for the DBLP and CSX datasets.

Citeseer[2]: Similarly, the Citeseer dataset (which we call CSX) had 1.3 million publication titles, and we preprocessed the dataset into a CSV format as in [109]. One difference from DBLP is that the "misc" field also contained an abstract and URLs to the publications' references, making the records larger than those in the DBLP dataset on average. As shown in Figure 6.22, the schema for CSX was basically identical to the one for DBLP.

Dataset Scaling: We scaled both the DBLP and CSX dataset up to 10 times their original size using the method in [109]. The scaling technique aims to increase the datasets in such a way that the result size of a set-similarity self-join on the "title" attribute grows linearly with the scale factor (assuming the Jaccard function with a 0.8 similarity threshold).

6.5.3.2 Synthetic Data: SimBench

In order to precisely study the performance effects of different similarity-predicate selectivities on large data, we also generated a synthetic dataset called "SimBench" whose schema is shown in Figure 6.23. We created a family of name fields, name_1E1, name_1E3 and name_1E5, intended to test queries based on edit distance. We created their field values in such a way that a selection query using edit distance with a threshold of 2 would yield exactly 10^1, 10^3, and 10^5 results, respectively. To provide this property, we generated the field values with the following process. First, we downloaded a dataset from the US Census

[2]http://citeseerx.ist.psu.edu/about/metadata

Bureau[3] containing 5,494 frequent first names and 88,799 frequent last names. We then created a set of *base* names by randomly choosing a first and last name from that dataset, also giving a middle name or initial with a 15% chance each (but never both). Based on the number of names in the US Census dataset and the number of records we generated, the chance of two *base* names being similar to each other was very low. Next, for each base name we generated the desired number of matching *variants* by performing 1-2 random edit operations to the base name. As a result, we could precisely control the number of matches within edit distance 2 for any of the base names. Table 6.4 shows an excerpt of data values generated in such a way for the "name_1E1" field. We generated the base value "danyelle rikard" (shown in the first row of the table) by picking a random first and last name from the US Census dataset. The other nine variant values in the table were generated from the base value using random edit operations as described above.

```
create type SimBenchType as closed {
  id: int32,
  name_1E1: string,
  name_1E3: string,
  name_1E5: string,
  status_1E1: string,
  status_1E3: string,
  status_1E5: string
}
```

Figure 6.23: ASTERIX schema definition for the SimBench dataset.

In a similar fashion, we produced a family of status fields, status_1E1, status_1E3 and status_1E5 to experiment with queries based on the Jaccard similarity. The goal was to generate status strings such that a selection query using Jaccard and a similarity threshold of 0.9 yields a desired number of results. Each status was a string containing 10-19 words separated by space. To generate the base statuses we preprocessed the dictionary file from a Linux OS in `/usr/share/dict/words` by converting all words to lower case and removing duplicate entries. From that dictionary containing approximately 235,000 distinct words we randomly

[3]http://www.census.gov/genealogy/www/data/1990surnames/names_files.html

id	name_1E1	status_1E1
70790311	danyelle rikard	h ce b mycelian a abelmoschus [...] bee
96504119	danyelle likard	h ce b mycelian a abelmoschus [...] bee swooper
67442653	danyaelle rikard	h ce b mycelian a abelmoschus [...] bee polynucleate
71246130	danyelle irikad	h ce b mycelian a abelmoschus [...] bee oligoclase
156859241	zdanyelle rizkard	h ce b mycelian a abelmoschus [...] bee counterdrive
161098748	danyelhle rikard	h ce b mycelian a abelmoschus [...] bee battue
178240606	danyeyle rkard	h ce b mycelian a abelmoschus [...] bee caramelization
214369902	danrlle rikard	h ce b mycelian a abelmoschus [...] bee silverbelly
181042773	danyplle rikard	h ce b mycelian a abelmoschus [...] bee gulpin
57683253	danylle rikard	h ce b mycelian a abelmoschus [...] bee flandowser
...	...	...

Table 6.4: Excerpt of data generated for the SimBench dataset showing only the "id", "name_1E1", and "status_1E1" fields. The first row contains base values from which the corresponding variant values in the remaining nine rows are derived.

selected words using on a Zipf distribution to form a base status string (picking 10-19 words for each status string). We employed a Zipf distribution (skew exponent of 1) to emulate the skewed frequency of words in real-world text data. Next, we created status variants by adding another random word to a base status string, ensuring that the Jaccard similarity of a base status and a status variant (when tokenized into words) is no less than 0.9. Table 6.4 contains a few example values for the "status_1E1" field generated in such a fashion. First, we generated a base value shown in the first row of the table, and then we added random words at the end of the base value to generate its variants. Note that we use "[...]" in the table as a placeholder for words that we omitted for ease of exposition.

Finally, we assigned the "id" field by randomly choosing, without repetition, a number from the universe of positive signed 32-bit integers. Note that we could not simply create increasing id numbers because ASTERIX uses clustered primary indexes to store datasets. To avoid distorting our measurements due to such clustering, we wanted to decluster the base fields and their variants in the primary index with our id assignment scheme.

6.5.3.3 Dataset Statistics

The statistics of datasets used in the experiments are summarized in Table 6.5. It shows the number of records, the raw size as text (CSV format), and the average record size in text for the datasets. Since the scaling of the DBLP and CSX datasets is almost perfectly linear, we only list the scale factors 1 and 10 for brevity.

Dataset	#Records	Raw Size	Avg Record Size
DBLPx1	1,268,017	0.32GB	267 Bytes
DBLPx10	12,680,170	3.18GB	269 Bytes
CSXx1	1,385,532	1.78GB	1382 Bytes
CSXx10	13,855,320	17.93GB	1390 Bytes
SimBench	1,907,770,415	522.18GB	294 Bytes

Table 6.5: Statistics of datasets.

Table 6.6 contains statistics on the fields of interest to our experiments. In particular, it shows the minimum, maximum, and average number of word tokens ("Length" in the table) for the "title" and "status" fields that we used for Jaccard-based queries, as well as those statistics on the "name" field that we used for edit-distance-based queries.

Dataset	Field	Min Length	Max Length	Avg Length
DBLPx1	title	0	93	9
DBLPx10	title	0	93	9
CSXx1	title	0	112	8
CSXx10	title	0	112	8
SimBench	name_1E1	3	38	16
	name_1E3	3	36	16
	name_1E5	4	33	16
	status_1E1	10	20	15
	status_1E3	10	20	15
	status_1E5	10	20	15

Table 6.6: Field Statistics. For the title and status fields, "length" refers to the number of word tokens. For the name fields "length" is the number of characters.

6.5.4 Primary and Secondary Index Sizes

To better understand the performance results, we measured the sizes of all primary indexes (for datasets) and secondary indexes. The primary-index sizes are shown Table 6.7. The DBLP and CSX dataset are still relatively small even at scale factor 10, and their primary indexes easily fit into main memory. To also measure the I/O performance of our solutions, we designed the SimBench dataset following conventional wisdom [34] that its primary indexes should be approximately five times the size of available memory on one node (12GB). Each node stores four partitions, so the total size of all SimBench primary indexes on a single node is 4*15.12GB=60.48GB.

Dataset	Avg Size Per Partition	Total Size
DBLPx1	9.75MB	0.38GB
DBLPx10	94.30MB	3.68GB
CSXx1	47.49MB	1.87GB
CSXx10	476.28MB	18.65GB
SimBench	15.12GB	609.39GB

Table 6.7: Primary-index sizes on 10 nodes with 40 partitions (4 partitions per node).

Tables 6.8 and 6.9 summarize the sizes of the secondary inverted index used in our experiments. Table 6.8 shows the sizes of the regular secondary inverted indexes, and Table 6.9 of the corresponding length-partitioned versions. The tables show the sizes of the B-Tree dictionaries ("Dict") and the inverted-list files ("Inv. Lists") separately to highlight that the main difference between the regular and fuzzy indexes is the size of their dictionaries.

6.5.5 Set-Similarity Selection Queries

To study the performance of similarity-selection queries, we generated a workload of 100 queries for each of the fields in the SimBench dataset (a total of 6 workloads with 100 queries each). We used the AQL template shown in Figure 6.24(a) to create the edit-

		Avg Size Per Partition			Total Size		
Dataset	**Field**	**Dict**	**Inv. Lists**	**Total**	**Dict**	**Inv. Lists**	**Total**
CSXx1	title	1.25MB	1.38MB	2.26MB	0.05GB	0.05GB	0.10GB
CSXx10	title	3.38MB	13.75MB	17.12MB	0.13GB	0.54GB	0.67GB
SimBench	name_1E1	1.00MB	4.05GB	4.05GB	40.00MB	162.11GB	162.15GB
	name_1E3	1.00MB	4.05GB	4.05GB	40.00MB	162.10GB	162.14GB
	name_1E5	1.00MB	4.05GB	4.05GB	40.00MB	162.13GB	162.17GB
	status_1E1	8.12MB	3.24GB	3.25GB	325.00MB	129.75GB	130.07GB
	status_1E3	8.12MB	3.27GB	3.27GB	325.00MB	130.63GB	130.94GB
	status_1E5	8.12MB	3.26GB	3.27GB	325.00MB	130.45GB	130.76GB

Table 6.8: Regular secondary inverted-index sizes.

		Avg Size Per Partition			Total Size		
Dataset	**Field**	**Dict**	**Inv. Lists**	**Total**	**Dict**	**Inv. Lists**	**Total**
CSXx1	title	3.25MB	1.38MB	4.62MB	0.13GB	0.05GB	0.18GB
CSXx10	title	12.97MB	13.75MB	26.72MB	0.51GB	0.54GB	1.04GB
SimBench	name_1E1	16.21MB	4.05GB	4.07GB	0.63GB	162.11GB	162.75GB
	name_1E3	16.12MB	4.05GB	4.07GB	0.63GB	162.10GB	162.73GB
	name_1E5	15.50MB	4.05GB	4.07GB	0.63GB	162.13GB	162.73GB
	status_1E1	89.62MB	3.24GB	3.33GB	3.50GB	129.75GB	133.25GB
	status_1E3	82.50MB	3.27GB	3.35GB	3.22GB	130.62GB	133.85GB
	status_1E5	82.00MB	3.26GB	3.34GB	3.20GB	130.45GB	133.65GB

Table 6.9: Length-partitioned secondary inverted-index sizes.

distance-based queries on the "name" fields. We formed a concrete query by replacing the placeholders "%F", "%C", "%T" with the desired name field, constant search argument, and similarity threshold, respectively. Similarly, Figure 6.24(b) shows the template for Jaccard-based queries on the "status" fields. For each query, we picked a random base variant (see Section 6.5.3.2) as its constant search argument to guarantee a certain number of results. For example, a selection query on the "status_1E3" field based on a Jaccard-similarity threshold of 0.9 yields exactly 1000 results by design. Recall that we assigned the primary-key values (for the "id" field) in the SimBench dataset in a random fashion to minimize the performance impact of id-based clustering in the primary index. At the same time, our primary-key assignment scheme also minimizes the inter-query and intra-query effects of caching pages in the BufferCache. Further, we ran the AQL query in Figure 6.25 between workloads (100

queries each); it requires a full scan on each partition, and clears the BufferCache of useful pages from the previous workload.

```
for $s in dataset('SimBench')
where edit-distance($s.%F, '%C') <= %T
return $s
```

(a) Edit-Distance Selection Query

```
for $s in dataset('SimBench')
where similarity-jaccard($s.%F, '%C') >= %T
return $s
```

(b) Jaccard Selection Query

Figure 6.24: AQL query templates for generating similarity-selection queries.

We do not report the performance of similarity selection queries on the DBLP and CSX datasets due to their small size. Those datasets, including their secondary indexes, easily fit into main memory (see Tables 6.7, 6.8, and 6.9), and we found that the response times of indexed similarity-selection queries were consistently in the order of a few hundred milliseconds. Further, the performance of similarity selection queries can be reasonably inferred from our experiments on similarity joins in Section 6.5.6 because an indexed nested-loops similarity join is essentially an optimized series of indexed selection queries.

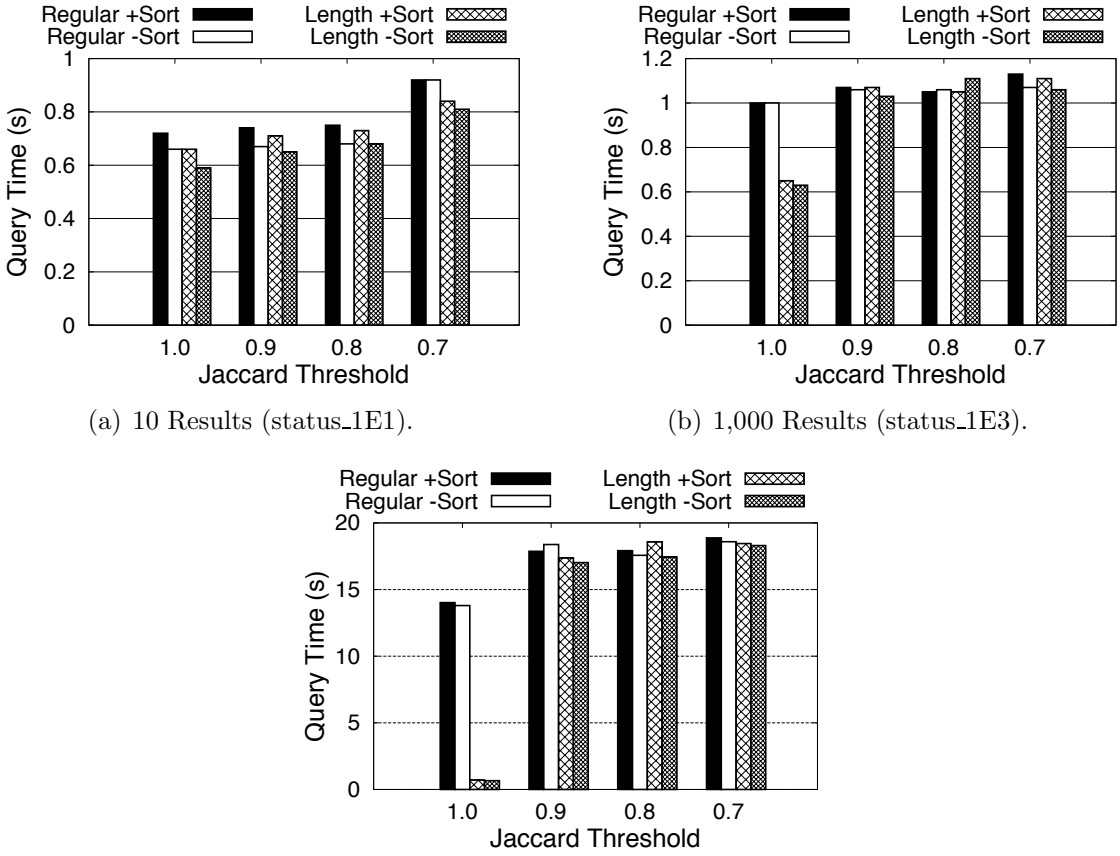
6.5.5.1 Selection Queries Based on Jaccard

Figure 6.26 shows the results of the performance experiments on Jaccard-based selection queries. To produce the results we ran the 100-query workloads corresponding to the fields "status_1E1", "status_1E3", and "status_1E5" of the SimBench dataset using different Jaccard thresholds. We plotted the average query response time per workload. Recall that

```
count(for $s in dataset('SimBench') return $s)
```

Figure 6.25: AQL query for clearing the BufferCache from a previous workload.

we generated the SimBench data in such a way that such a selection query would yield exactly 10, 1,000, and 100,000 results for each "status" field, respectively. We enabled and disabled sorting of the primary keys after the secondary-index search (see Section 6.3.5.1), denoted by "+Sort" and "-Sort". We tested both the regular and length-partitioned versions of our inverted index, denoted by "Regular" and "Length", to measure their performance difference. Perhaps surprisingly, the index and query-plan variants yielded almost identical performance, explained as follows.



(a) 10 Results (status_1E1).

(b) 1,000 Results (status_1E3).

(c) 100,000 Results (status_1E5).

Figure 6.26: Performance results of selection queries based on Jaccard.

Sorting vs. Not Sorting: A simple analysis reveals that sorting the primary keys after the secondary-index lookup is unlikely to be fruitful. For example, consider the most promising case for sorting, namely the 100,000-result bars in Figure 6.26(c). We expect that most

of the execution time for such a query would be spent in fetching the disk pages for the primary-index lookups. There were at least 100,000 such lookups, but there could have been more due to false positives. Because ASTERIX was running on 40 partitions, each partition fetched roughly 100,000/40=2500 primary keys, assuming a uniform distribution of data items and a low false-positive rate. The primary index on each partition was 15.12GB in size (see Table 6.7), which translates to approximately 123,863 128KB disk pages. It seems unlikely that sorting 2500 records for finding them among 123,863 pages will yield any significant savings. That being said, sorting the primary keys did not hurt performance either, because 2500 primary-key integers easily fit into the 256MB of memory allocated to the sort operators (see Table 6.3). This result suggests to always enable sorting to protect against queries with an unanticipated high number of false positives or results. To perform a sanity check on our results, we can take the above calculations further and assume a conservative disk-seek time of 8ms which results in 2500*8ms=20s for a selection query with 100,000 results. The conservatively estimated 20s per query matches our results in the figures. We conclude that ASTERIX performed as expected for Jaccard-based selection queries returning 100,000 results. On the other hand, based on the above reasoning, we would expect the performance of queries with 10 and 1000 results to be proportional to those with 100,000 results. For example, we might estimate that queries returning 1,000 results should be 100,000/1,000=100 times faster than those returning 100,000 results. The reason why our experiments for 1,000 query results did not match this estimation is that, at query latencies below 1 second, much of the time is spent on "base" activities such as query compilation/optimization and sending/instantiating Hyracks operator pipelines. The queries returning 10 results responded in approximately 700 milliseconds, and those returning 1,000 results finished in approximately 1 second.

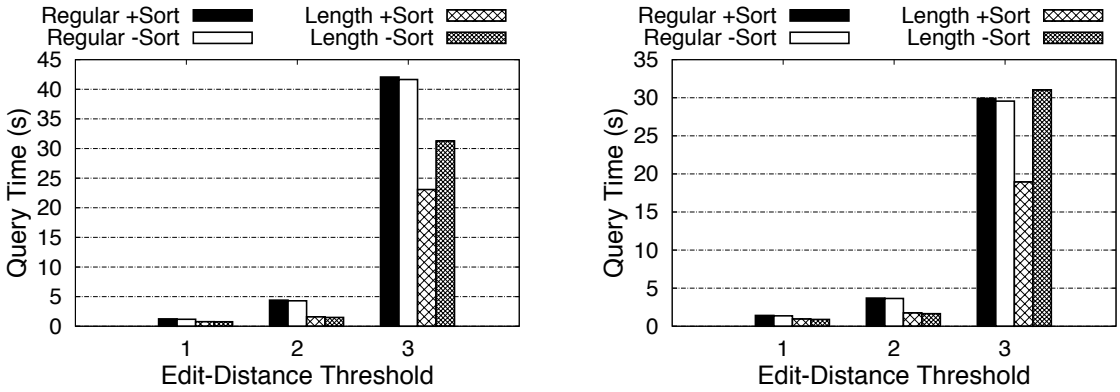
Regular vs. Length-Partitioned Index: In general, the main advantage of a length-partitioned index over its regular version is that it reduces the number of false-positive answers. The reason why the length-partitioned indexes used in Figure 6.26 showed little

improvement over the corresponding regular ones is that the "status" field values in the SimBench dataset were quite dissimilar to each other. As a result, the T-Occurrence filter (see Section 2.4) could already eliminate most of the candidate answers, and length filtering became useless. The fact that decreasing the Jaccard threshold had little effect on query performance also supports this hypothesis. As we decreased the Jaccard threshold the query response times stayed roughly constant, regardless of whether we used regular of length-partitioned inverted indexes. The only exceptions where a length-partitioned index was significantly beneficial was for a Jaccard threshold of 1.0 in Figures 6.26(b) and 6.26(c), which is explained as follows. Remember that we generated the status' variant values by adding a word to a base variant, and that we used base values as search arguments in our selection queries here. Since a Jaccard threshold of 1.0 implies that every result must have an identical number of set elements, the length filter could eliminate all variant values for a given base value, resulting in a dramatic performance improvement.

6.5.5.2 Selection Queries Based on Edit Distance

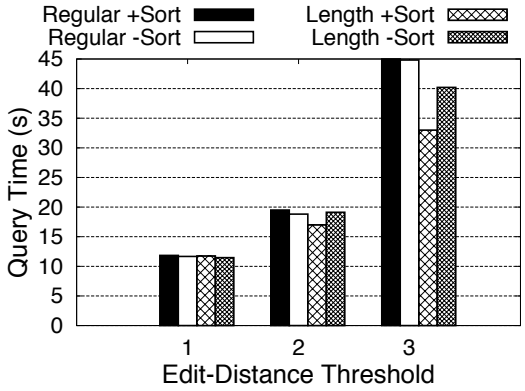
Figure 6.27 shows the results of the performance experiments on edit-distance-based selection queries. For each of the "name" fields we ran its corresponding 100-query workload using varying edit-distance thresholds, and plotted the average query response times. As before, we enabled and disabled sorting of the primary keys ("+Sort" and "-Sort"), and tested both regular indexes and length-partitioned indexes ("Regular" and "Length").

Our results show the following general trends. First, the query response time increased with higher edit-distance thresholds for all of the "name" fields. Second, the length-partitioned indexes improved the query performance by 2-3 times in many cases. For example, in Figure 6.27(a) for an edit-distance threshold of 2 the regular index answered queries in 4.4s, whereas the length-partitioned index did so in 1.5s. Third, sorting the primary keys had little effect on query performance, with a few exceptions to be discussed shortly.



(a) 10 Results (name_1E1).

(b) 1,000 Results (name_1E3).



(c) 100,000 Results (name_1E5).

Figure 6.27: Performance results of selection queries based on edit distance.

Jaccard vs. Edit Distance: One striking difference between the results in Figure 6.27 and those for the Jaccard-based queries in Figure 6.26 is that the queries based on edit distance were generally much slower. The main reason for this difference is that there were significantly fewer distinct 3-grams used to create the indexes on the "name" fields than there were distinct dictionary words used to generate the "status" field values. Intuitively, the 3-gram sets of the "name" field values were more similar to each other than were the word-token sets of the "status" field values. Therefore, there was a comparatively higher chance that two "name" strings satisfy the T lower bound (see Section 2.4), resulting in more false positives as compared to the Jaccard-based queries on the "status" fields. This fact also explains why the length-partitioned indexes proved more useful for edit-distance-based

queries. The additional length filter could significantly reduce the number of false positives that survived the T-Occurrence filter. The results in Figure 6.27(c) further validate this hypothesis. For example, the "name_1E5" field contained 100-times fewer base variants than the "name_1E3" field. As a result, the relative number of false positives was smaller for queries on the "name_1E5" field than on the "name_1E3" field. Therefore, the length-partitioned indexes provided comparatively little benefit on the "name_1E5" field.

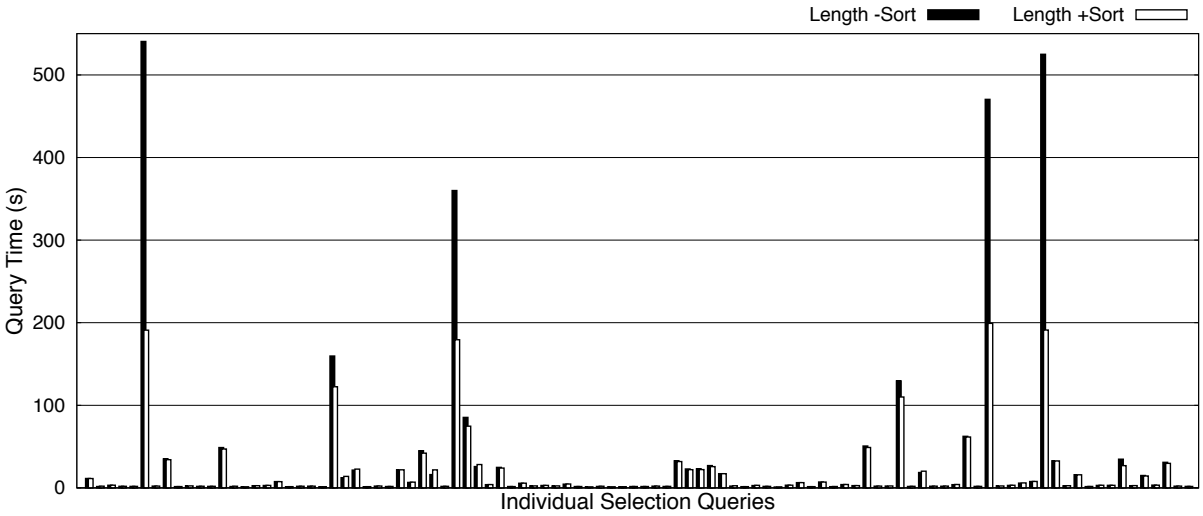


Figure 6.28: Response times of individual selection queries on the "name_1E3" field for an edit-distance threshold of 3, a length-partitioned index, and the non-sorting (left bar) and sorting (right bar) plan variants.

Sorting vs. Not Sorting: As shown in Figure 6.27, sorting the primary keys after the secondary-index search yielded no significant performance improvement for the edit-distance thresholds 1 and 2. On the other hand, we observed a drop in response times due to sorting for an edit-distance threshold of 3 on length-partitioned indexes. Notice that this improvement in performance was consistent across all the "name" fields. Consider Figure 6.28 for an explanation. We plotted the individual response times of the selection-query workload corresponding to Figure 6.27(b) for an edit-distance threshold of 3, a length-partitioned index, and the non-sorting (left bar) and sorting (right bar) plan variants. We see that most of the queries completed quickly, and only a few queries were very slow. Interestingly, some

queries were even slower than a scan-based execution strategy, which took about 200s for this setting. Those expensive queries had short search strings resulting in a small T lower bound that was ineffective for pruning candidate answers. For example, if $T = 1$ for a particular query then its candidate result set is the union of all inverted lists relevant to the query. In general, the response time for such queries with many candidates is dominated by fetching the full data items from the primary index for removing false positives. Sorting the primary keys yielded significant improvements for those few, expensive queries because it improved BufferCache hit rates during the primary-index searches. On the other hand, notice that sorting did not improve the performance of queries on regular inverted indexes on the same workload with those few many-candidate queries (e.g., see Figure 6.27(b)). The reason for this behavior is that the results obtained from regular inverted indexes are often "almost" sorted by primary key already, due to the merge-sort-like nature of the underlying inverted-list-merging algorithm (however, there is no guarantee of ordering). We use a similar list merging algorithm for the length-partitioned inverted indexes; however, we process one length partition at a time, delivering an overall "less" sorted candidate result set.

6.5.6 Set-Similarity Join Queries

To study the performance of similarity join queries we generated self-join queries for each field in the SimBench dataset using the AQL templates shown in Figure 6.29. The placeholders "%F" and "%T" represent the desired field and similarity threshold, respectively. Notice the additional conditions on the "id" field in the "where" clause. The first condition on "id" is intended to precisely control the number of tuples coming from the probe side of the join (see Section 6.3.3) by restricting the tuples to a range of primary-key values. The placeholder "%PKL" represents the lower bound and "%PKH" the upper bound of the desired primary-key value range. As the SimBench dataset is stored in a primary index on the "id" field, this range condition is rewritten into a primary-index search during query optimization.

The second condition on the "id" field removes duplicate pairs of similar data items, which can occur because we are doing a self join and the similarity functions are symmetric. We wrapped the join queries into a "count()" function because ASTERIX currently writes query results to a single output partition and we wanted to exclude the possibly long time to write large result files from our measurements.

```
count(
  for $x in dataset('SimBench')
  for $y in dataset('SimBench')
  where edit-distance($x.%F, $y.%F) <= %T
        and $y.id >= %PKL and $y.id < %PKH
        and $y.id < $x.id
  return {'x': $x, 'y': $y}
)
```

(a) Edit-Distance Join Query

```
count(
  for $x in dataset('SimBench')
  for $y in dataset('SimBench')
  where similarity-jaccard($x.%F, $y.%F) >= %T
        and $y.id >= %PKL and $y.id < %PKH
        and $y.id < $x.id
  return {'x': $x, 'y': $y}
)
```

(b) Jaccard Join Query

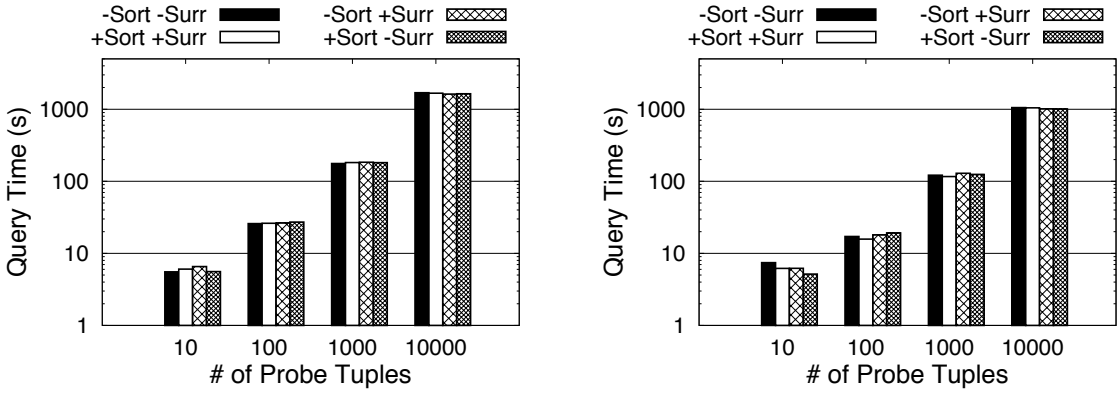
Figure 6.29: AQL query templates for generating similarity-join queries.

For each field in the SimBench dataset we generated four workloads designed to test a certain probe-side size. Specifically, we generated workloads for the probe-side sizes 10, 100, 1,000, and 10,000. Each workload consisted of five different queries with the same probe-side size. To create five such queries, we manually selected five non-overlapping ranges of primary-key values that each return exactly the desired number of tuples for the probe side. We chose non-overlapping ranges to minimize the effects of inter-query caching of disk pages. When running our experiments we again ran the query in Figure 6.25 between workloads to clear the BufferCache of useful pages from a previous workload.

In our experiments we focused on self-joins with a varying but reasonably small probe side for two reasons. First, conventional wisdom on equi joins [35] suggests that indexed nested-loop joins are typically only suitable when the probe side of the join is small. Second, based on our current and previous experiences with parallel set-similarity joins [109], running very large joins would take a very long time. For example, in [109] a set-similarity join of two datasets with a few million records took hours on the same cluster, and our SimBench dataset has orders of magnitudes more records (1.9 billion).

6.5.6.1 Join Queries Based on Jaccard

Figure 6.30 shows the results of the performance experiments on Jaccard-based join queries on the "status_1E1" field using a fixed Jaccard threshold of 0.9. We tested both regular (Figure 6.30(a)) and length-partitioned indexes (Figure 6.30(b)), and enabled and disabled sorting of the primary keys ("+Sort" and "-Sort") and the surrogate-based join-plan variant from Section 6.3.5.2 ("+Surr" and "-Surr").



(a) Regular Index.

(b) Length-Partitioned Index.

Figure 6.30: Performance results of Jaccard-based join queries with a threshold of 0.9 on the "status_1E1" field.

For both index types, the query runtime increased linearly with the size of the probe side. The reason is that the cost of processing a single tuple via the secondary-to-primary index

search plan was roughly constant, and therefore, the overall running time of the join was governed mainly by how many index lookups were performed by each index. Recall that in our indexed nested-loops join plan the probe side is broadcast to all partitions, so the number of index lookups performed by each partition in parallel is exactly the size of the probe side. Compared to the regular indexes, the length-partitioned indexes answered queries up to 60% faster due to the superior pruning of candidate answers via the additional length filtering.

Query Plan Variants: We measured no significant performance difference between the tested query-plan variants. As before, sorting the primary keys here did not help performance because there were too few of them, but again, and for the same reason as before sorting did not hurt performance, either. The surrogate-based join-plan variant ("+Surr") yielded almost identical response times to the regular plan ("-Surr"). Based on the relatively small probe-side sizes and the SimBench's average record size of 300 bytes, it is easy to see that the network cost of broadcasting the probe side should be negligible compared to the disk I/O during the subsequent index lookups. For example, take the probe-side size of 10,000 tuples. Assuming a uniform distribution of data items among the 40 partitions, each partition stores 10,000/40=250 such data items. To perform the broadcast, each partition sends all data items to all other partitions, shipping a total of 250*40*40*300bytes=115MB over the network (sometimes over the loopback device). This network data transfer should take one second on our Gigabit connection, assuming the switch can handle all concurrent transfers with a full Gigabit/s bandwidth. Further, since our query plan was pipelined, the sender side of the broadcast exchWange only sent the next batch of tuples once they had been consumed by the upstream plan (the secondary-to-primary index lookups). Within the same reasoning, we were surprised that the additional top-level join of the surrogate-based plan did not negatively affect the response time of queries. Again, this can be explained by the relatively small probe-side size. Each partition of the top-level hash join could easily fit all data items of its probe side into its hash table (to which we had allocated 256MB of buffer space). As a result, the overhead of the top-level join was minimal. Choosing between

the regular and surrogate-based join plans should ideally be done in a cost-based fashion. However, our results suggest a simple heuristic might be to always use the surrogate-based plan because of its potential benefits and minimal overhead.

Varying Jaccard Thresholds: To measure the performance of Jaccard-based join queries with varying Jaccard thresholds we ran the self-join workload corresponding to the "status_1E1" field and 10,000 probe tuples. We enabled the sorting of primary keys and disabled the surrogate-based plan. The results based on using a regular and length-partitioned index are shown in Figure 6.31. As expected, the query response time increased steadily as we decreased the Jaccard threshold for both types of indexes because of additional false positives and bigger results. Our experiments also showed that length-filtering improved the response time by up to 60%, but it became less effective with a lower Jaccard threshold. Surprisingly, for a threshold of 0.7, the length-partitioned index performed slightly worse than the regular one, explained as follows. The values of the "status_1E1" field had an average of 15 word tokens (see Table 6.6). Based on a Jaccard threshold of 0.7 length filtering bounds the size of any set similar to such an average query set of 15 tokens to the range [10, 22]. As a result, length filtering was completely ineffective (on average) for a threshold of 0.7 and even added overhead, because the processing of *all* length-partitions inside the index is more expensive as compared to processing a single large partition (for the regular indexes).

6.5.6.2 Join Queries Based on Edit Distance

Our experimental results for edit-distance-based join queries on the "name_1E1" field with a varying probe-side size are shown in Figure 6.32. We used an edit-distance threshold of 2 and ran the workloads against regular and length-partitioned indexes, denoted by "Regular" and "Length", respectively. We always enabled the sorting of primary keys because in the edit-distance-based join plan from Section 6.3.3.1 a pipeline-breaking operator such as sorting is required in the index-based branch before the union operator (the left input of the union

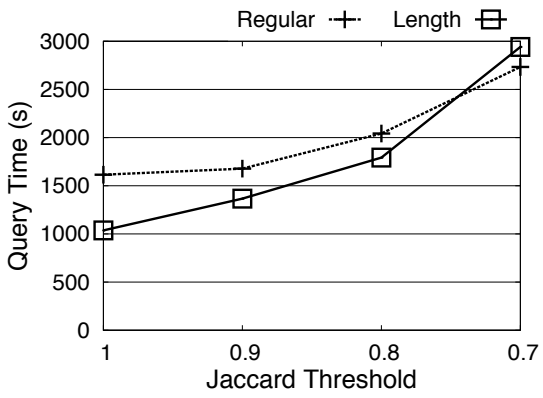


Figure 6.31: Performance results of Jaccard-based join queries with varying thresholds on the "status_1E1" field with 10,000 probe tuples.

in Figure 6.12). Intuitively, our query plan must guarantee that all inputs of the union operator are executed at the same time (i.e., in the same Hyracks stage [20]). Otherwise, there will be a deadlock when the union operator waits for tuples from one of its inputs that has not been scheduled for execution yet. Since the nested-loop join on the right input branch of the union is a pipeline-breaking operator (two phases: build and join), we also need a corresponding pipeline-breaking operator on the left input branch of the union. This deadlock behavior is a limitation of our current fully-pipelined implementation of the union operator which consumes batches of tuples from its inputs in a round-robin fashion and then immediately forwards them to its consuming operator.

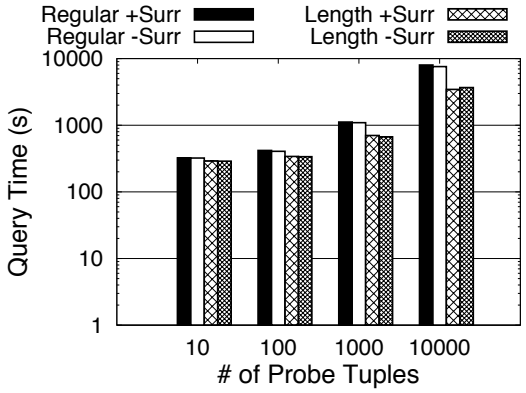


Figure 6.32: Performance results of edit-distance-based join queries with a threshold of 2 on the "name_1E1" field.

The results in Figure 6.32 show surprisingly slow response times for small probe-side sizes such as 10 and 100, explained as follows. Due to the nested-loop join in the "panic" branch of the query plan (see Figure 6.12), we must always execute a full scan of the index side of the join (the side with 1.9 billion data items). We must execute this scan even if none of the tuples from the probe side qualify as a panic case, i.e., their T lower bound is always greater than zero. The reason is that in Hyracks we currently cannot "advise" the scheduler to not execute the scan based on the number of input tuples from the build side of the "panic" nested-loop join. Even so, avoiding the scan is conceptually possible and we leave this improvement for future work as it requires significant changes. This mandatory full scan dominated the overall query time of the 10 and 100 probe-input sizes, and it is the reason why their times did not differ by much. As we increased the probe-input size to 1,000 and 10,000, the time for this scan became less dominant, but it still added a sizable overhead. In general, a scan of the SimBench dataset took 200-300s. As the number of probe tuples increased, we also saw more significant time differences between the regular and length-partitioned indexes. For example, at a probe-input size of 10,000, a length-partitioned index answered queries about twice as fast as its regular counterpart due to a reduction in false positives. As with the Jaccard-based join queries, we did not see a significant difference between the surrogate-based plans ("+Surr") and the regular plans ("-Surr") because of the relatively small probe-side sizes.

Varying Edit-Distance Thresholds: Figure 6.33 reports the experimental results on joins with varying edit-distance thresholds. We fixed the probe-side size to 1,000 and ran the self-join workload for the "name_1E1" field using a regular and length-partitioned index with the non-surrogate plan variant. The length-partitioned index answered queries with an edit-distance threshold of 1 and 2 approximately 40% faster than the regular version. For an edit-distance threshold of 3 we even measured an improvement of 3x over the regular index. The increase in response times with higher edit distances is due to the combination of the following two factors. First, increasing the edit-distance threshold by 1 decreases the

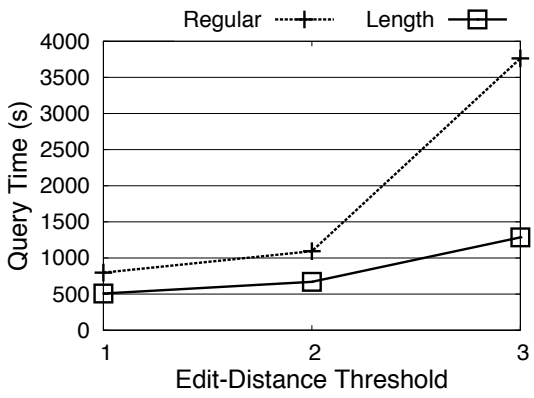


Figure 6.33: Performance results of join queries using varying edit-distance thresholds on the "name_1E1" field with 1,000 probe-side tuples.

T-occurrence lower bound by 3 because we are using 3-grams. Second, "name" strings are short (an average of 16 characters) and there are few distinct 3-grams relative to the total number of "name" values in the dataset. Those two factors combined lead to significantly less pruning by the inverted lists, resulting in a dramatic increase in false positives at an edit-distance threshold of 3. This reduced pruning power of the T lower bound also explains why the length-partitioned index was so effective, namely because length filtering could eliminate many irrelevant candidates early.

6.5.7 Comparison with Unindexed Set-Similarity Join

We compared our indexed nested-loops join strategy with an existing parallel algorithm for answering set-similarity queries [109] that is also implemented in ASTERIX. Our goal was to understand their performance differences to help the query optimizer decide between those two plan alternatives. We focus the comparison on set-similarity joins based on Jaccard, because the implementation of the parallel set-similarity join from [109] currently only supports Jaccard. That existing solution does not use secondary indexes, and therefore, we call it the "*NO-IX*" (no index) approach. We refer to our approach as the "*IX-NL*" (indexed

nested loops) approach.

The *NO-IX* method uses a three-staged query plan illustrated by Vernica's original example in Figure 6.34 which we borrowed from [109]. Next, we will briefly describe the query plan of the *NO-IX* method. Please refer to [109] for a more details. Suppose we wanted to join two ASTERIX datasets "DBLP" and "CSX" based on the Jaccard similarity of their title fields (schemas shown in Figure 6.22). The first stage of the *NO-IX* technique computes the (approximate) global frequency of all set elements in the join attribute. To obtain these element frequencies it performs a scan on either dataset (DBLP in the figure) followed by a grouped aggregation whose result is then sorted in ascending order of the element frequencies. This map of element frequencies is broadcast to all nodes, where it serves as a global ordering of all set elements. In the second stage, we re-partition and replicate some fields of both DBLP and CSX using the prefix filtering idea (see Section 5.2.1). For each data item in DBLP and CSX we compute its prefix-element set based on the element-frequency ordering and then send that data item to one node for each element in its prefix set (using hash partitioning). For example, if a data item has 5 elements in its prefix set, then we replicate it to 5 different nodes by hashing on the prefix-set elements. To reduce network costs during this step, the *NO-IX* approach only transfers the primary-key field and the join-attribute field to the other nodes (similar to our surrogate-based plan from Section 6.3.5.2). This prefix-based partitioning strategy ensures that two potentially similar data items sharing at least one element in their prefix set will end up on the same node. Next, we execute a local set-similarity join on each node in parallel on the re-partitioned data items from DBLP and CSX. As a result, each node has a list of primary-key pairs whose data items satisfy the similarity join condition. In the third and final step, we construct the complete join result by fetching the data items corresponding to those primary keys of both sides of the join. Two equi-joins are needed for this final step. To summarize, the *NO-IX* approach requires multiple full scans of both join sides, repartitioning of both sides, and multiple groupings and equi-joins.

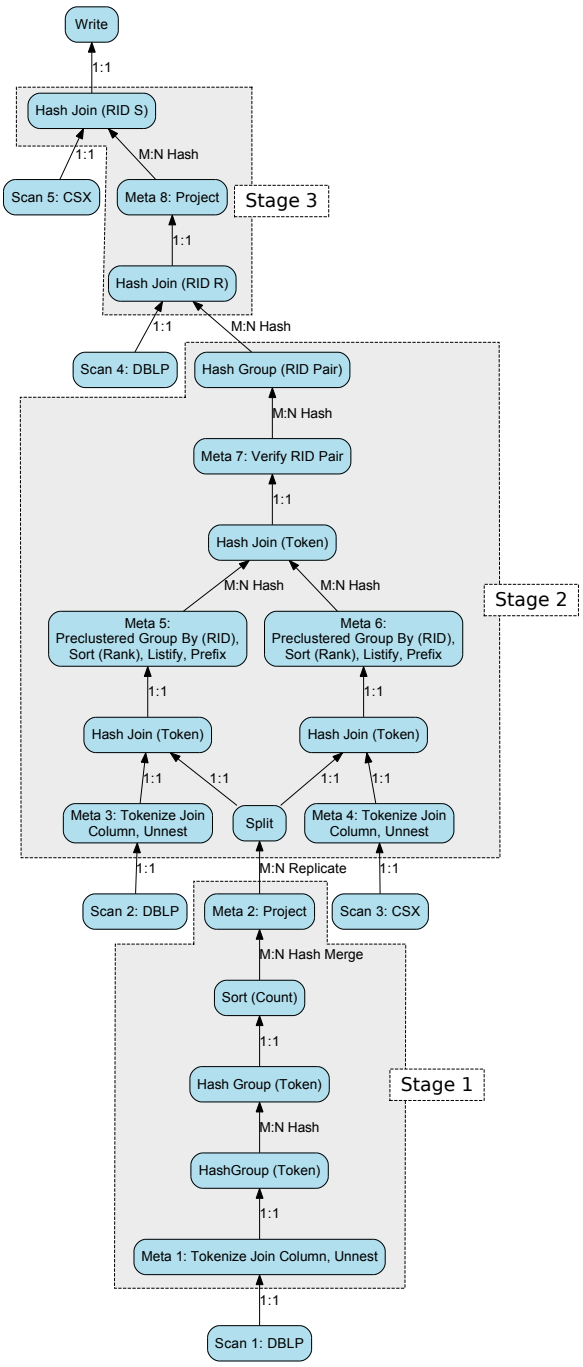
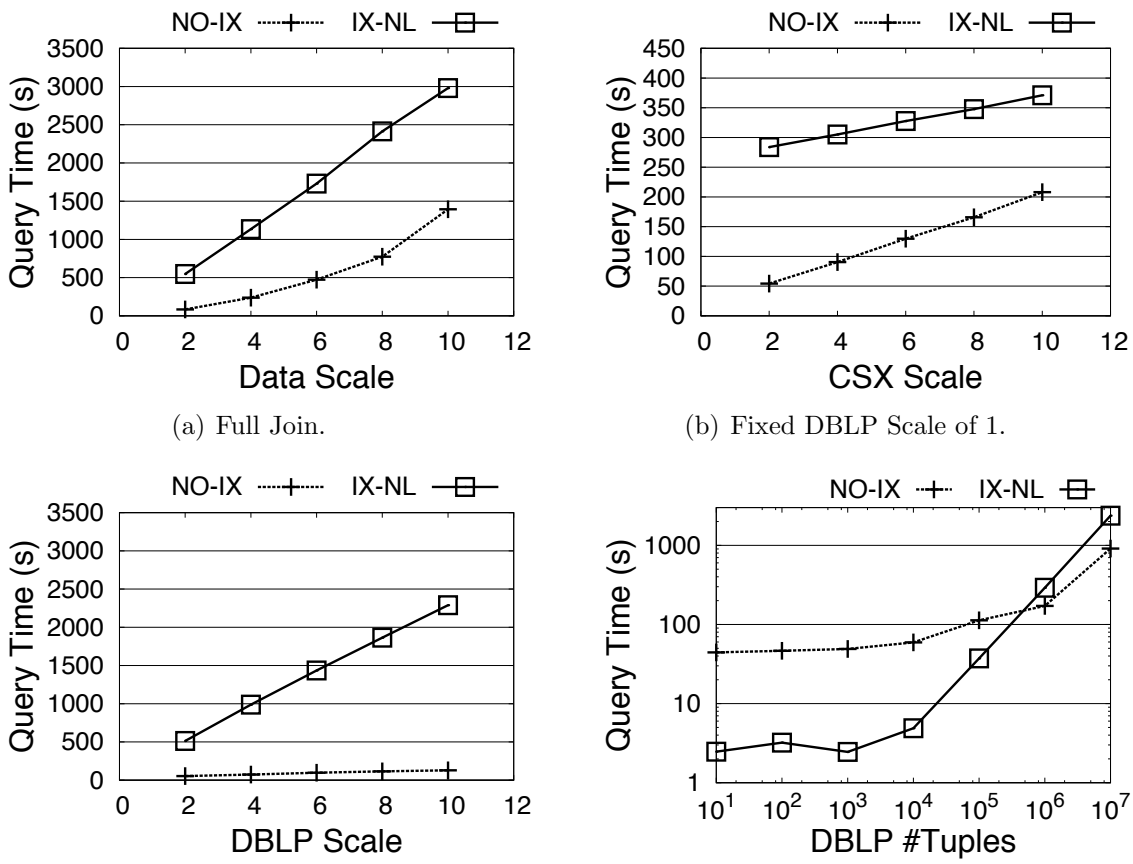


Figure 6.34: Hyracks physical plan for a set-similarity join between the DBLP and CSX datasets using the *NO-IX* join method. Image source: R. Vernica [109], p. 119.

6.5.7.1 Set-Similarity Join Between the DBLP and CSX Datasets

To establish a baseline comparison we repeated the experiments from [109] on the DBLP and CSX datasets with scale factors from 1 to 10 (see Section 6.5.3.1). We ran set-similarity joins between DBLP and CSX on the word-tokens of their title field using Jaccard and a threshold of 0.9. For the *IX-NL* approach we created a length-partitioned secondary index on CSX's title field. Note that these datasets and the secondary index easily fit into main memory, even at a scale factor of 10 (see Tables 6.7 and 6.9). Therefore, we disabled the sorting of primary keys in the *IX-NL* approach because it only added overhead and no benefit. On the other hand, we did enable the surrogate-based plan for *IX-NL* as it performed approximately 20% faster than the regular plan, since it reduced data copies in the critical path of the query plan (secondary-to-primary index lookups).

Figure 6.35 summarizes the results of our experiments on the DBLP and CSX datasets. Figure 6.35(a) shows the response times of joining the two datasets with increasing scale factors. We observed that the *NO-IX* method consistently outperformed the *IX-NL* approach. Due to its divide-and-conquer strategy, the *NO-IX* method effectively reduced the overall join into smaller local joins with *both* join-inputs reduced in size. Compare this with the *IX-NL* method where each data partition performed an index search for every data item from the DBLP dataset. In some sense, using multiple machines only reduced one side of the join (the CSX side) in the index-based solution. Based on this observation it is easy to explain *IX-NL*'s linear increase in the response time as we scaled up the dataset sizes. It took roughly a constant amount of time for a single secondary-to-primary index search, hence the overall time of our indexed nested-loops join was governed by how many such index lookups were performed. On the other hand, the increase in the query times of the *NO-IX* approach seemed to accelerate with higher scale factors, due to its inherent quadratic complexity as already reported in [109]. Intuitively, the main performance difference between the *NO-IX* and the *IX-NL* approaches can be summarized as follows. For two datasets R and S, the



(a) Full Join.

(b) Fixed DBLP Scale of 1.

(c) Fixed CSX Scale of 1.

(d) Varying Probe-Side Size.

Figure 6.35: Comparison between the *NO-IX* and *IX-NL* approaches for joining the DBLP and CSX datasets on their titles' word tokens using Jaccard and a threshold of 0.9.

cost of a *NO-IX* join is $C_1 * |R| * |S|$, and the cost for a *IX-NL* join is $|R| * C_2$, assuming an index on S, where C_1 and C_2 are data-dependent constant factors. This cost model is certainly not accurate but it does explain the general trends. For example, one inaccuracy stems from the assumption that secondary-to-primary index lookups have a constant cost, which may not be true, of course. However, as we will see shortly, the cost of such index lookups grows rather slowly as more data is indexed.

To gain a deeper understanding of the performance effects of the individual join-input sizes we ran the following experiments. First, we fixed the DBLP dataset (the probe side) to a scale factor of 1 and increased the size of the CSX dataset (the index side). The results

are shown in Figure 6.35(b). We see that the query time of *IX-NL* grew relatively slower than that of *NO-IX*. This result demonstrates that the secondary index on the CSX dataset scaled well when indexing more data, and also reinforces our analysis that the dominating factor of the *IX-NL* approach is the probe-side size. As predicted by our simple cost model, the response time of *NO-IX* grew roughly linearly since only one side of the join increased in size. We also tested the other extreme scenario where we fixed the CSX dataset to a scale of 1 and varied the size of the DBLP dataset, the results of which are plotted in Figure 6.35(c). Again, the response time of *NO-IX* grew linearly, but as we increased the probe-side of the join the *IX-NL* join became much more expensive. More importantly, the response time of *IX-NL* grew almost exactly proportional to the size of the DBLP dataset.

Our analysis and experiments so far suggest that *IX-NL* should outperform *NO-IX* if the probe-side of the join is small enough. To test this idea, we used the DBLP and CSX datasets with a scale factor of 10 and varied the probe-side size. We added range predicates on the primary-key "id" field of DBLP to precisely control how many data items would flow from it into the join. Figure 6.35(d) shows the results as we varied the probe-side size from 10 to 10 million. As expected, the *IX-NL* was significantly faster for small probe-side sizes, but it was eventually outperformed by the *NO-IX* approach. We attribute this behavior to the fact that *NO-IX* is a scan-based solution that does not exploit the skewness of the join-input sizes, whereas the index-based plan does. As a result, the index-based plan needed to examine many fewer data items for small probe-side sizes.

6.5.7.2 Set-Similarity Self-Join of the SimBench Dataset

We also compared the *NO-IX* and *IX-NL* approaches on the SimBench dataset. We did several self-joins on its "status_1E1" field using a Jaccard threshold of 0.9. As before, we used range conditions on the primary-key field to generate five queries with non-overlapping primary-key ranges per probe-side size. For the *IX-NL* approach we created a length-

partitioned secondary index on the "status_1E1" field, enabled sorting of the primary keys, and disabled the surrogate-based plan. Because the SimBench dataset is large, we expected sorting to reduce the I/O costs of the primary-index lookups, and in the worst case, to have no impact on performance as already demonstrated in Section 6.5.6.1. Figure 6.36 shows the results of those experiments. As with the DBLP and CSX datasets, we observed that *IX-NL* outperformed *NO-IX* for relatively small probe-side sizes. Notice that the query times were generally higher and their differences more pronounced than on the DBLP and CSX dataset due to the disk I/O required for processing the joins on the large SimBench dataset. The results verify our previous observations: the query time of the *IX-NL* approach increased proportionally to the probe-side size, and the query time of the *NO-IX* solution grew more slowly.

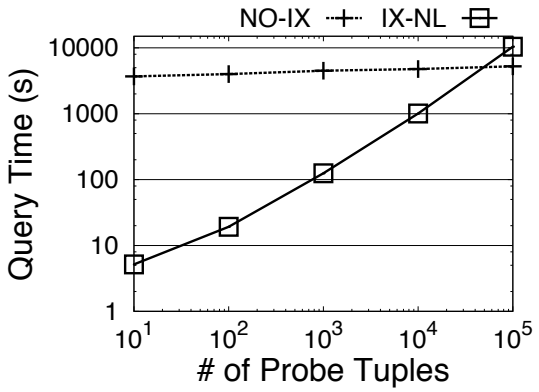


Figure 6.36: Comparison between the *NO-IX* and *IX-NL* approaches for self-joining the SimBench dataset on its "status_1E1" field using Jaccard and a threshold of 0.9.

6.5.8 Summary of Experiments

We summarize our experimental study on indexed similarity selection and join queries with the following observations.

- Selection queries with common similarity thresholds (e.g., a Jaccard of 0.9 or an edit

distance of 2) responded within 5 seconds on large data, and often even within 1 second.

- The indexed nested-loops query plan for answering similarity joins has a cost proportional to that of the probe-side size.

- The indexed join plan is significantly faster than the unindexed join solution from [109] for small probe-size sizes. For large probe-side sizes the index-based plan becomes expensive and is outperformed by the unindexed solution. These observations complement existing findings for equi-join strategies [34], and show that analogous tradeoffs based on the relative join-input sizes exist for similarity joins.

- The length-partitioned indexes often outperform their regular counterparts by a factor of two to three. However, for scenarios where length filtering is ineffective in pruning false positives, a length-partitioned index could be slightly slower than a regular index.

- Sorting the primary keys resulting from secondary index searches is sometimes very beneficial, and almost never hurtful. We conclude that using sorting by default may be a good choice to guard against unexpectedly large primary-key sets.

- The surrogate-based join-plan variant was neither beneficial nor detrimental to performance in most of our experiments. The reason is that we only tested moderate probe-side sizes on a small 10-node cluster, and therefore, broadcasting the complete records of the probe side was not a performance bottleneck. On the other hand, the additional top-level join to resolve the surrogates did not negatively affect performance, either. Hence, the surrogate-based plan may provide a good default option to protect against unforseen broadcasting costs.

Chapter 7

Conclusions and Future Work

7.1 Conclusions

Answering similarity selection and join queries is important in many applications where inconsistencies and errors occurring in queries and data need to be dealt with appropriately. Due to the diversity of applications that could benefit from similarity-querying capabilities, it stands to reason that one single solution cannot adequately meet the requirements of all possible applications. In particular, for each application we expect different data characteristics and sizes, similarity functions, hardware constraints, query workloads, and performance requirements. In this thesis, we have developed several index-based methods to efficiently answer similarity queries with different application desiderata in mind. Our solutions have focused on a core framework for answering queries based on widely applicable similarity functions such as edit distance and Jaccard. Our contributions are summarized as follows.

In Chapter 3 we presented a specialized solution for the application of DNA sequence mapping. The problem is to map a large set of small DNA snippets (reads) to a known DNA reference sequence while tolerating a few mismatches based on Hamming or edit distance.

The following properties characterize this application domain. First, the alphabet of strings is limited to the 4 characters, A, C, T, and G. Second, the reference sequence is typically fixed, and therefore, the indexing strategy need not deal with online updates. Third, modern high-throughput sequencing machines can rapidly produce large amounts of DNA reads, and therefore, the speed of read-mapping solutions is very important. We have developed a gram-based read-mapping package called Hobbes that improves upon existing solutions. Hobbes implements two novel techniques that yield substantial performance gains: an optimized gram-selection procedure for reads, and a cache-efficient filter for pruning candidate mappings. We have systematically tested the performance of Hobbes on both real and simulated data with read lengths varying from 35 to 100 base pairs, and compared its performance with several state-of-the-art read-mapping programs, including Bowtie, BWA, mrsFast, and RazerS. We have shown that Hobbes is faster than all these read mapping programs while maintaining high mapping quality.

In Chapter 4 we developed two techniques for reducing the size of a q-gram inverted index for answering global-alignment queries. Such q-gram inverted indexes are known for their large size relative to the data they index. Some applications may require very low response times for similarity queries on large data, and therefore it is desirable to hold the supporting indexes entirely in memory. To make more efficient use of available hardware resources it is important to reduce the memory footprint of the indexes supporting similarity queries. We have first studied how to adopt existing inverted-list compression techniques to our setting. Then, we presented two independent approaches for reducing the size of a q-gram inverted index to a given amount of space while retaining efficient query processing. The first approach is based on the idea of discarding some of the inverted lists. The second approach minimizes redundancy in the index by combining some of the correlated inverted lists. For both approaches we have studied the effects of such compression on queries, and developed cost-based algorithms for constructing an index based on a given memory constraint and a given query workload. Our experimental results showed that we could reduce the index size

by up to 60% without sacrificing query response times, and that often existing approaches do not perform as well.

In Chapter 5 we studied external-memory solutions for answering global-alignment similarity queries. Our approach is well-suited for scenarios where indexes and data are assumed to be on disk, e.g., in database management systems. Storing the data items and the inverted lists on disk dramatically changes the costs of answering similarity queries, so this setting presents new tradeoffs and allows novel optimizations. We have proposed a multi-level inverted index to minimize the I/O costs of answering queries. We have presented a new storage layout for an inverted index that is optimized for answering similarity queries, and we have shown how to efficiently construct such an index with limited buffer space. The new layout is based on partitioning the data items such that at query time only a few partitions must be considered, significantly reducing the number of false positives. To answer queries efficiently, we have developed a cost-based algorithm that balances the I/O costs of accessing inverted lists and candidate answers. This algorithm is based on the intuition that very long inverted lists are expensive to retrieve and offer little pruning power, and hence, we should avoid reading them depending on the query. Our experiments on real data have shown that the combination of our techniques outperforms a standard inverted index as well as a recently proposed tree-based index called BED-Tree.

In Chapter 6 we discussed how we integrated some of our solutions into the ASTERIX parallel database management system. ASTERIX is a shared-nothing database system for storing, indexing, and querying large quantities of semi-structured data. It is geared towards the new breed of "Big Data" applications which aim to derive knowledge from the vast quantities of data being generated on a daily basis on the web, for example, in blogs, social networks, online communities, click streams, etc. Data gathered from said sources is bound to contain noise and inconsistencies, since many users publish their data informally resulting in abbreviated keywords and misspellings. Our goal was to add indexed support for efficiently

answering similarity selection and join queries based on Jaccard and edit distance. We have presented how to express set-similarity queries in the declarative ASTERIX query language, and have introduced data-definition language constructs to create secondary indexes for optimizing such similarity queries. To use secondary indexes in query processing, we have developed the corresponding rewrite rules used in the ASTERIX optimizer. Finally, we have presented our implementation of regular and length-partitioned inverted indexes based on ASTERIX's Log-Structured-Merge framework. We have experimented with different query-plan alternatives on large-scale data, and have compared our index-based join approach with an existing unindexed approach. We have verified that the standard practice of sorting the primary keys resulting from a secondary-index search is sometimes beneficial and never hurtful. As conventional wisdom states, we have experimentally found that the index-based similarity join outperforms an unindexed join solution if the side of the join without the index is relatively small.

7.2 Future Work

In Chapter 6 we have focused on answering set-similarity queries with secondary indexes that are co-partitioned with their primary index, often referred to as *local* secondary indexes. As a consequence of this partitioning, both selection and join queries using such a local secondary index must be executed on all of the secondary-index partitions, because no partition pruning based on the secondary-index condition is possible. For selection queries this strategy may be inefficient because it always involves all nodes with secondary-index partitions, even if the query only asks for a few data items. Assuming a large cluster, we expect many nodes to return an empty result after probing their secondary-index partitions, which is wasteful for such highly selective queries. For index-based join queries we have seen that the dominating cost of answering them is the probe-side size because every secondary-index

partition must process all tuples from the probe side. This broadcast-based join execution is a direct consequence of using local secondary indexes. A promising direction for future research is to investigate alternative partitioning strategies for secondary indexes based on the secondary-index key, often referred to as *global* indexes. For example, we could use the length filtering idea from Section 5.2.1 to partition a secondary index across nodes based on the length of the indexed data items. This partitioning strategy would enable similarity selection and join queries to prune secondary-index partitions based on length filtering. For selection queries such a global-index partitioning could reduce the number of nodes involved in answering a query significantly. For join queries this partitioning strategy eliminates the need to broadcast the probe side. Instead, each data item from the probe side is only sent to those relevant nodes that could contain answers based on the length filter. One drawback of global secondary indexes is that the primary keys resulting from searching a particular secondary-index partition may not be on the same node, hence, answering queries using global indexes incurs extra network costs to fetch the results from their primary-index partitions. We believe that studying different global-index partitioning strategies and comparing their performance to local secondary indexes is an interesting direction for future research.

Our experiments on ASTERIX suggest that we could improve the performance of index-based similarity queries by removing false positives before accessing the primary index. The following are a few interesting approaches that deserve attention. In Section 5.3.1 we introduced a separate index called "dense index" exactly for this purpose. However, since ASTERIX is targeting insert-heavy use cases it is not immediately obvious whether maintaining such an additional index is cost effective, and the corresponding tradeoff between insert and search performance deserves to be investigated. Another direction that does not rely on a separate index is to augment an existing inverted index (see Section 6.4) with positional information. That is, for each token we could have an inverted list that contains the primary key of the data item containing that token and the position at which the token

appeared in the corresponding field in the data item. During an index search we can use those positions and our dictionary B-Tree to reconstruct the original field values and perform the similarity-verification step to remove false positives. Thus, we could avoid accessing the primary index for irrelevant answers. However, adding positions to inverted lists increases their size and complicates the index search algorithm. Also, in order to reconstruct the original data values the dictionary B-Tree must use the original token values and not hashed tokens. It is an interesting question whether the "dense index" solution or the positional approach (or neither) have merit in practice.

Another possible future direction is to study parallel multi-field similarity joins, perhaps using indexes. In applications such as record linkage, we aim to fuse data from different sources by matching the real-world entities from both sources and combining their data values into a single master dataset. Typically, the matching of such entities is more effective if several attributes are considered during the matching process. For example, suppose we wanted to fuse two customer datasets having fields such as "name", "address", "phone", etc. One way to match such data items is to consider two items similar if (1) their "name" fields are within a certain edit distance, and (2) the Jaccard similarity of their "address" fields exceeds a certain threshold. Alternatively, we could combine the similarity of the "name" field and that of the "address" field of two data items using a combination function, and then put a threshold on the combined similarity value. There are several simple ways to execute such multi-attribute similarity joins, e.g., we could pick one of the attributes and then do a single-attribute similarity join followed by a selection to handle the other attributes' conditions. However, it might be more efficient to jointly consider all or a subset of the attributes of the similarity-join condition, e.g., by combining their token sets and computing a joint T-Occurrence threshold. Similarly, it is an interesting question whether we could create a composite secondary index to directly answer such multi-attribute similarity queries and in which scenarios such a composite index is superior to intersection-based strategies using several individual secondary indexes.

Bibliography

[1] http://yh.genomics.org.cn/.

[2] ftp://ftp.ddbj.nig.ac.jp/ddbj_database/dra/fastq/DRA000/DRA000222/DRX000359/.

[3] ftp://ftp.ddbj.nig.ac.jp/ddbj_database/dra/fastq/DRA000/DRA000222/DRX000360/.

[4] ftp://ftp-trace.ncbi.nih.gov/1000genomes/ftp/data/HG00096/sequence_read/.

[5] http://bio.math.berkeley.edu/eXpress/index.html.

[6] Press release: Gartner says master data management software revenue to grow 21 percent in 2012, http://www.gartner.com/it/page.jsp?id=1886314. *Gartner Group.*

[7] *Oracle Text, An Oracle Technical White Paper*, 2007. http://www.oracle.com/technology/products/text/pdf/11goracletexttwp.pdf.

[8] *Fuzzy Search in IBM DB2 9.5*, 2008. http://publib.boulder.ibm.com/infocenter/db2luw/v9r5/index.jsp?topic=/com.ibm.db2.luw.admin.nse.topics.doc/doc/t0052178.html.

[9] J. M. Abowd and L. Vilhuber. The sensitivity of economic statistics to coding errors in personal identifiers. *Journal of Business and Economic Statistics*, 23(2):pp. 133–152, 2005.

[10] C. Alkan, J. M. Kidd, T. Marques-Bonet, G. Aksay, F. Antonacci, F. Hormozdiari, J. O. Kitzman, C. Baker, M. Malig, O. Mutlu, S. C. Sahinalp, R. A. Gibbs, and E. E. Eichler. Personalized copy-number and segmental duplication maps using next-generation sequencing. *Nature Genetics*, 41(10):1061–1067, 2009.

[11] S. Altschul, W. Gish, W. Miller, E. Myers, and D. Lipman. Basic local alignment search tool. *Journal of molecular biology*, 215(3):403–410, 1990.

[12] V. N. Anh and A. Moffat. Inverted index compression using word-aligned binary codes. *Inf. Retr.*, 2005.

[13] A. Arasu, V. Ganti, and R. Kaushik. Efficient exact set-similarity joins. In *VLDB*, pages 918–929, 2006.

[14] ASTERIX, http://asterix.ics.uci.edu.

[15] M. J. Bauer, A. J. Cox, and D. J. Evers. ELANDv2 - fast gapped read mapping for illumina reads. In *ISMB*. ISCB, 2010.

[16] R. J. Bayardo, Y. Ma, and R. Srikant. Scaling up all pairs similarity search. In *WWW*, 2007.

[17] A. Behm, V. R. Borkar, M. J. Carey, R. Grover, C. Li, N. Onose, R. Vernica, A. Deutsch, Y. Papakonstantinou, and V. J. Tsotras. Asterix: towards a scalable, semistructured data platform for evolving-world models. *Distributed and Parallel Databases*, 29:185–216, 2011.

[18] A. Behm, S. Ji, C. Li, and J. Lu. Space-constrained gram-based indexing for efficient approximate string search. In *ICDE*, 2009.

[19] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, July 1970.

[20] V. Borkar, M. J. Carey, R. Grover, and N. O. aand Rares Vernica. Hyracks: A flexible and extensible foundation for data-intensive computing. In *ICDE*, april 2011.

[21] L. Boytsov. Super-linear indices for approximate dictionary searching. In *Proceedings of the 5th international conference on Similarity Search and Applications*, SISAP'12, pages 162–176, Berlin, Heidelberg, 2012. Springer-Verlag.

[22] S. Buettcher and C. L. A. Clarke. Index compression is good, especially for random access. In *CIKM*, 2007.

[23] S. Burkhardt and J. Kärkkäinen. Better filtering with gapped q-grams. *Fundam. Inf.*, 56(1,2):51–70, 2002.

[24] M. Burrows and D. Wheeler. A block sorting lossless data compression algorithm. *Technical Report 124. Palo Alto, CA: Digital Equipment Corporation*, 25, 1994.

[25] S. Chaudhuri, K. Ganjam, V. Ganti, R. Kapoor, V. Narasayya, and T. Vassilakis. Data cleaning in Microsoft SQL Server 2005. In *SIGMOD*, 2005.

[26] S. Chaudhuri, K. Ganjam, V. Ganti, and R. Motwani. Robust and efficient fuzzy match for online data cleaning. In *SIGMOD Conference*, 2003.

[27] S. Chaudhuri, V. Ganti, and L. Gravano. Selectivity estimation for string predicates: Overcoming the underestimation problem. In *ICDE*, pages 227–238, 2004.

[28] S. Chaudhuri, V. Ganti, and R. Kaushik. A primitive operator for similarity joins in data cleaning. In *ICDE*, 2006.

[29] S. Chaudhuri and R. Kaushik. Extending autocompletion to tolerate errors. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, SIGMOD '09, pages 707–718, New York, NY, USA, 2009. ACM.

[30] D. Chung, P. Kuan, B. Li, R. Sanalkumar, K. Liang, E. Bresnick, C. Dewey, and S. Keleş. Discovering transcription factor binding sites in highly repetitive regions of genomes with multi-read analysis of chip-seq data. *PLoS Computational Biology*, 7(7):e1002111, 2011.

[31] D. W. Collins and T. H. Jukes. Rates of transition and transversion in coding sequences since the human-rodent divergence. *Genomics*, 20(3):386 – 396, 1994.

[32] L. T. Corporation. Life technologies introduces the benchtop ion proton sequencer; designed to decode a human genome in one day for $1,000. `http://www.lifetechnologies.com/us/en/home/about-us/news-gallery/press-releases/2012/life-techologies-itroduces-the-bechtop-io-proto.html.html`, 2012. [Online; accessed 16-Jan-2012].

[33] S. B. David R. Bentley, H. P. Swerdlow, G. P. Smith, J. Milton, C. G. Brown, K. P. Hall, D. J. Evers, and et. al. Accurate whole genome sequencing using reversible terminator chemistry. *Nature*, 456:53–50, november 2008.

[34] D. J. DeWitt. The wisconsin benchmark: Past, present, and future. In *The Benchmark Handbook*. Morgan Kaufmann, 1993.

[35] D. J. DeWitt, J. F. Naughton, and J. Burger. Nested loops revisited. In *In Proceedings of the Symposium on Parallel and Distributed Information Systems*, pages 216–226. Morgan-Kaufman, Inc, 1993.

[36] A. Döring, D. Weese, T. Rausch, and K. Reinert. Seqan an efficient, generic c++ library for sequence analysis. *BMC Bioinformatics*, 9(1):11, 2008.

[37] W. E. Eckerson. Data quality and the bottom line. *The Data Warehousing Institute*, 2002.

[38] P. Elias. Universal codeword sets and representations of the integers. *Information Theory, IEEE Transactions on*, 21(2):194–203, Mar 1975.

[39] P. Ferragina and G. Manzini. An experimental study of an opportunistic index. In *Proceedings of the twelfth annual ACM-SIAM symposium on Discrete algorithms*, pages 269–278. Society for Industrial and Applied Mathematics, 2001.

[40] P. Fogla and W. Lee. q-gram matching using tree models. *IEEE Trans. Knowl. Data Eng.*, 18(4):433–447, 2006.

[41] T. Friedman. Magic quadrant for data quality tools. *Gartner Group*, 2012.

[42] Z. Galil and G. F. Italiano. Data structures and algorithms for disjoint set union problems. *ACM Comput. Surv.*, 23(3):319–344, 1991.

[43] G. Graefe. Modern b-tree techniques. *Found. Trends databases*, 3(4):203–402, Apr. 2011.

[44] L. Gravano, P. G. Ipeirotis, H. V. Jagadish, N. Koudas, S. Muthukrishnan, and D. Srivastava. Approximate string joins in a database (almost) for free. In *VLDB*, pages 491–500, 2001.

[45] A. Guttman. R-trees: a dynamic index structure for spatial searching. In *Proceedings of the 1984 ACM SIGMOD international conference on Management of data*, SIGMOD '84, pages 47–57, New York, NY, USA, 1984. ACM.

[46] F. Hach, F. Hormozdiari, C. Alkan, F. Hormozdiari, I. Birol, E. E. Eichler, and S. C. Sahinalp. mrsfast: a cache-oblivious algorithm for short-read mapping. *Nature Methods*, 7(8):576–577, 2010.

[47] M. Hadjieleftheriou, A. Chandel, N. Koudas, and D. Srivastava. Fast indexes and algorithms for set similarity selection queries. In *ICDE*, 2008.

[48] M. Hadjieleftheriou and D. Srivastava. Approximate string processing. *Foundations and Trends in Databases*, 2(4):267–402, 2011.

[49] M. Hadjieleftheriou, X. Yu, N. Koudas, and D. Srivastava. Hashed samples: Selectivity estimators for set similarity selection queries. In *VLDB*, 2008.

[50] F. Hao, J. Daugman, and P. Zielinski. A fast search algorithm for a large fuzzy database. *Information Forensics and Security, IEEE Transactions on*, 3(2):203–212, june 2008.

[51] M. R. Henzinger. Finding near-duplicate web pages: a large-scale evaluation of algorithms. In *SIGIR*, pages 284–291, 2006.

[52] T. C. Hoad and J. Zobel. Methods for identifying versioned and plagiarized documents. *JASIST*, 54(3):203–215, 2003.

[53] B. Hore, H. Hacigümüs, B. R. Iyer, and S. Mehrotra. Indexing text data under space constraints. In *CIKM*, pages 198–207, 2004.

[54] D. Huffman. A Method for the Construction of Minimum-Redundancy Codes. *Proceedings of the IRE*, 40(9):1098–1101, Sept. 1952.

[55] E. Hunt, M. P. Atkinson, and R. W. Irving. A database index to large biological sequences. In *In VLDB*, pages 139–148, 2001.

[56] P. Indyk and R. Motwani. Approximate nearest neighbors: Towards removing the curse of dimensionality. In *STOC Conference*, 1998.

[57] H. V. Jagadish, R. T. Ng, and D. Srivastava. Substring selectivity estimation. In *PODS*, pages 249–260, 1999.

[58] C. Jermaine, E. Omiecinski, and W. G. Yee. The partitioned exponential file for database storage management. *The VLDB Journal*, 16(4):417–437, Oct. 2007.

[59] S. Ji, G. Li, C. Li, and J. Feng. Efficient interactive fuzzy keyword search. In *Proceedings of the 18th international conference on World wide web*, WWW '09, pages 371–380, New York, NY, USA, 2009. ACM.

[60] Y. Ji, Y. Xu, Q. Zhang, K. Tsui, Y. Yuan, C. Norris Jr, S. Liang, and H. Liang. Bmmap: Bayesian mapping of multireads for next-generation sequencing data. *Biometrics*, 2011.

[61] L. Jin and C. Li. Selectivity estimation for fuzzy string predicates in large data sets. In *VLDB*, 2005.

[62] P. Jokinen, J. Tarhio, and E. Ukkonen. A comparison of approximate string matching algorithms. *Softw. Pract. Exper.*, 26(12):1439–1458, 1996.

[63] P. Jokinen and E. Ukkonen. Two algorithms for approxmate string matching in static texts. In A. Tarlecki, editor, *Mathematical Foundations of Computer Science 1991*, volume 520 of *Lecture Notes in Computer Science*, pages 240–248. Springer Berlin Heidelberg, 1991.

[64] L. Kaufman and P. Rousseeuw. *Finding Groups in Data: an introduction to cluster analysis*. John Wiley and Sons, New York, 1990.

[65] M.-S. Kim, K.-Y. Whang, J.-G. Lee, and M.-J. Lee. n-Gram/2L: A space and time efficient two-level n-gram inverted index structure. In *VLDB*, 2005.

[66] M. Kircher and J. Kelso. High-throughput dna sequencing - concepts and limitations. *BioEssays*, 32(6), 2010.

[67] D. Kossmann. The state of the art in distributed query processing. *ACM Comput. Surv.*, 32(4):422–469, Dec. 2000.

[68] N. Koudas, S. Sarawagi, and D. Srivastava. Record linkage: similarity measures and algorithms. In *SIGMOD Conference*, pages 802–803, 2006.

[69] P. Krishnan, J. S. Vitter, and B. R. Iyer. Estimating alphanumeric selectivity in the presence of wildcards. In *SIGMOD Conference*, pages 282–293, 1996.

[70] B. Langmead, C. Trapnell, M. Pop, and S. L. Salzberg. Ultrafast and memory-efficient alignment of short dna sequences to the human genome. *Genome Biology*, 10, 2009.

[71] B. Langmead, C. Trapnell, M. Pop, and S. L. Salzberg. Ultrafast and memory-efficient alignment of short dna sequences to the human genome. *Genome Biology*, 10:r25, 2009.

[72] H. Lee, R. T. Ng, and K. Shim. Extending q-grams to estimate selectivity of string matching with low edit distance. In *VLDB*, pages 195–206, 2007.

[73] D. Lemire and L. Boytsov. Decoding billions of integers per second through vectorization. *CoRR*, abs/1209.2137, 2012.

[74] N. Lester, J. Zobel, and H. Williams. Efficient online index maintenance for contiguous inverted lists. *Inf. Process. Manage.*, 42(4), 2006.

[75] V. I. Levenshtein. Binary codes capable of correcting spurious insertions and deletions of ones. *Probl. Inf. Transmission*, 1965.

[76] C. Li, J. Lu, and Y. Lu. Efficient merging and filtering algorithms for approximate string searches. In *ICDE*, pages 257–266, 2008.

[77] C. Li, B. Wang, and X. Yang. VGRAM: Improving performance of approximate queries on string collections using variable-length grams. In *VLDB*, 2007.

[78] G. Li, D. Deng, J. Wang, and J. Feng. Pass-join: a partition-based method for similarity joins. *Proc. VLDB Endow.*, 5(3):253–264, Nov. 2011.

[79] H. Li. *wgsim - Read simulator for next generation sequencing, http://github.com/lh3/wgsim*, 2011.

[80] H. Li and R. Durbin. Fast and accurate short read alignment with burrows-wheeler transform. *Bioinformatics*, 25:1754–1760, 2009.

[81] H. Li and N. Homer. A survey of sequence alignment algorithms for next-generation sequencing. *Briefings in Bioinformatics*, 11(5):473–483, Sept. 2010.

[82] H. Li, J. Ruan, and R. Durbin. Mapping short dna sequencing reads and calling variants using mapping quality scores. *Genome research*, 18(11):1851, 2008.

[83] R. Li, C. Yu, Y. Li, T.-W. Lam, S.-M. Yiu, K. Kristiansen, and J. Wang. Soap2: an improved ultrafast tool for short read alignment. *Bioinformatics*, 25:1966–1967, 2009.

[84] Y. Li, A. Terrell, and J. M. Patel. Wham: a high-throughput sequence alignment method. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, SIGMOD '11, pages 445–456. ACM, 2011.

[85] H. Lin, Z. Zhang, M. Zhang, B. Ma, and M. Li. Zoom! zillions of oligos mapped. *Bioinformatics*, 24(21):2431, 2008.

[86] J. I. Maletic and A. Marcus. Data cleansing - a prelude to knowledge discovery. In *The Data Mining and Knowledge Discovery Handbook*, pages 21–36. 2005.

[87] A. Mazeika, M. H. Böhlen, N. Koudas, and D. Srivastava. Estimating the selectivity of approximate string queries. *ACM Trans. Database Syst.*, 32(2):12, 2007.

[88] M. D. McIllroy. Development of a spelling list. *IEEE Transactions on Communications*, 30(1):91–99, 1998.

[89] J. D. McPherson. *Nature Methods*, (11s):S2–S5, November 2009.

[90] C. Meek, J. M. Patel, and S. Kasetty. Oasis: An online and accurate technique for local-alignment searches on biological sequences, 2003.

[91] A. Metwally, D. Agrawal, and A. E. Abbadi. Detectives: detecting coalition hit inflation attacks in advertising networks streams. In *WWW*, pages 241–250, 2007.

[92] G. Meyers. A fast bit-vector algorithm for approximate string matching based on dynamic programming. *J. ACM*, 46:395–415, 1999.

[93] F. Miller, A. Vandome, and M. John. *Bernoulli Process*. VDM Publishing, 2010.

[94] A. Moffat and J. Zobel. Self-indexing inverted files for fast text retrieval. *ACM Trans. Inf. Syst.*, 14(4):349–379, 1996.

[95] C. Mohan and F. Levine. Aries/im: an efficient and high concurrency index management method using write-ahead logging. In *Proceedings of the 1992 ACM SIGMOD international conference on Management of data*, SIGMOD 1992, pages 371–380, New York, NY, USA, 1992. ACM.

[96] G. Navarro. A guided tour to approximate string matching. *ACM Comput. Surv.*, 33(1):31–88, 2001.

[97] H. B. Newcombe, J. M. Kennedy, S. J. Axford, and A. P. James. Automatic linkage of vital records. *Science (New York, N.Y.)*, 130(3381):954–959, Oct. 1959.

[98] D. Newkirk, J. Biesinger, A. Chon, K. Yokomori, and X. Xie. Arem: aligning short reads from chip-sequencing by expectation maximization. In *Research in Computational Molecular Biology*, pages 283–297. Springer, 2011.

[99] Z. Ning, A. J. Cox, and J. C. Mullikin. Ssaha: a fast search method for large dna databases. *Genome Res*, 11(10):1725–1729, 2001.

[100] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil. The log-structured merge-tree (lsm-tree). *Acta Inf.*, 33(4):351–385, June 1996.

[101] S. Rumble, P. Lacroute, A. Dalca, M. Fiume, A. Sidow, and M. Brudno. Shrimp: accurate mapping of short color-space reads. *PLoS computational biology*, 5(5):e1000386, 2009.

[102] M. Sahami and T. D. Heilman. A web-based kernel function for measuring the similarity of short text snippets. In *WWW*, pages 377–386, 2006.

[103] S. C. Sahinalp, M. Tasan, J. Macker, and Z. M. Özsoyoglu. Distance based indexing for string proximity search. In *ICDE*, pages 125–, 2003.

[104] S. Sarawagi and A. Kirpal. Efficient set joins on similarity predicates. In *SIGMOD Conference*, 2004.

[105] R. Sears and R. Ramakrishnan. blsm: a general purpose log structured merge tree. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD 2012, pages 217–228, New York, NY, USA, 2012. ACM.

[106] A. X. L. K. Shen and E. Torng. Large scale hamming distance query processing. In *ICDE*, 2011.

[107] A. Smith, Z. Xuan, and M. Zhang. Using quality scores and longer reads improves accuracy of solexa read mapping. *BMC bioinformatics*, 9(1):128, 2008.

[108] E. Ukkonen. Approximae string matching with q-grams and maximal matching. *Theor. Comut. Sci.*, 1:191–211, 1992.

[109] R. Vernica. Efficient processing of set-similarity joins on large clusters, 2011.

[110] R. Vernica, M. Carey, and C. Li. Efficient parallel set-similarity joins using MapReduce. In *SIGMOD Conference*, 2010.

[111] R. Vernica and C. Li. Efficient top-k algorithms for fuzzy search in string collections. In *KEYS*, pages 9–14, 2009.

[112] C. Wang, J. Wang, X. Lin, W. Wang, H. Wang, H. Li, W. Tian, J. Xu, and R. Li. Mapdupreducer: detecting near duplicates over massive datasets. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, SIGMOD 2010, pages 1119–1122, New York, NY, USA, 2010. ACM.

[113] J. Wang, J. Feng, and G. Li. Trie-join: efficient trie-based string similarity joins with edit-distance constraints. *Proc. VLDB Endow.*, 3(1-2):1219–1230, Sept. 2010.

[114] J. Wang, G. Li, and J. Fe. Fast-join: An efficient method for fuzzy token matching based string similarity join. In *Data Engineering (ICDE), 2011 IEEE 27th International Conference on*, pages 458–469, april 2011.

[115] J. Wang, G. Li, and J. Feng. Can we beat the prefix filtering?: an adaptive framework for similarity join and search. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD 2012, pages 85–96, New York, NY, USA, 2012. ACM.

[116] W. Wang, C. Xiao, X. Lin, and C. Zhang. Efficient approximate entity extraction with edit distance constraints. SIGMOD, pages 759–770, 2009.

[117] D. Weese, A.-K. Emde, T. Rausch, A. Döring, and K. Reinert. Razers-fast read mapping with sensitivity control. *Genome Research*, 19:1646–1654, 2009.

[118] W. E. Winkler. String comparator metrics and enhanced decision rules in the fellegi-sunter model of record linkage. In *Proceedings of the Section on Survey Research*, pages 354–359, 1990.

[119] W. E. Winkler. Overview of record linkage and current research directions. Technical report, Bureau of The Census, 2006.

[120] C. Xiao, W. Wang, and X. Lin. Ed-join: An efficient algorithm for similarity joins with edit distance constraints. In *VLDB*, 2008.

[121] C. Xiao, W. Wang, X. Lin, and J. X. Yu. Efficient similarity joins for near duplicate detection. In *WWW*, pages 131–140, 2008.

[122] C. Xiao, W. Wang, X. Lin, J. X. Yu, and G. Wang. Efficient similarity joins for near-duplicate detection. *ACM Trans. Database Syst.*, 36(3):15:1–15:41, Aug. 2011.

[123] X. Yang, B. Wang, and C. Li. Cost-based variable-length-gram selection for string collections to support approximate queries efficiently. In *SIGMOD Conference*, 2008.

[124] Z. Zhang, M. Hadjieleftheriou, B. C. Ooi, and D. Srivastava. Bed-tree: an all-purpose index structure for string similarity search based on edit distance. In *SIGMOD*, 2010.

[125] J. Ziv and A. Lempel. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory*, 24(5):530–536, 1978.

[126] J. Zobel and A. Moffat. Inverted files for text search engines. *ACM Comput. Surv.*, 38(2):6, 2006.

[127] M. Zukowski, S. Heman, N. Nes, and P. Boncz. Super-scalar RAM-CPU cache compression. 2006.