UCLA
UCLA Electronic Theses and Dissertations

**Title**
Highly Efficient String Similarity Search and Join over Compressed Indexes

**Permalink**
https://escholarship.org/uc/item/62w4m38r

**Author**
Xiao, Guorui

**Publication Date**
2023

Peer reviewed|Thesis/dissertation

eScholarship.org Powered by the California Digital Library
University of California

UNIVERSITY OF CALIFORNIA

Los Angeles

# Highly Efficient String Similarity Search and Join over Compressed Indexes

A thesis submitted in partial satisfaction
of the requirements for the degree
Master of Science in Computer Science

by

Guorui Xiao

2023

© Copyright by

Guorui Xiao

2023

ABSTRACT OF THE THESIS

Highly Efficient String Similarity Search and Join
over Compressed Indexes

by

Guorui Xiao
Master of Science in Computer Science
University of California, Los Angeles, 2023
Professor Carlo Zaniolo, Chair

String similarity search and string similarity join are essential operations in many fields. Existing solutions adopt a filter-and-verification framework and build inverted indexes based on generated signatures to prune dissimilar candidates. While existing solutions mainly focus on improving the query processing performance, little attention is paid to reducing the inverted indexes' memory consumption. In cases where the index size is larger than the memory, users must employ more expensive disk-based algorithms rather than in-memory ones. In this thesis, we propose a flexible framework CSS to reduce the index size and keep high query performance for string search and join applications. We give improved solutions for offline inverted list construction and introduce a new approach for the online construction of compressed inverted lists. Experimental results on large-scale datasets demonstrate that CSS can reduce memory consumption up to 5 times while having similar, or even better, query processing performance.

The thesis of Guorui Xiao is approved.

George Varghese

Todd D. Millstein

Carlo Zaniolo, Committee Chair

University of California, Los Angeles

2023

*To my parents Li Cao, Xiangsheng Xiao, and my girlfriend Yaqi Zhan who have always been there for me.*

TABLE OF CONTENTS

**1 Introduction** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **1**

**2 Preliminary** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **5**

2.1 String Similarity Search and Join . . . . . . . . . . . . . . . . . . . . 5

2.2 Inverted List Compression . . . . . . . . . . . . . . . . . . . . . . . 6

**3 A Unified Compression Framework** . . . . . . . . . . . . . . . . . . . . **9**

3.1 Existing Filtering Techniques . . . . . . . . . . . . . . . . . . . . . . 9

3.1.1 Count Filters . . . . . . . . . . . . . . . . . . . . . . . . . . 9

3.1.2 Prefix Filters . . . . . . . . . . . . . . . . . . . . . . . . . . 10

3.1.3 Position Filter . . . . . . . . . . . . . . . . . . . . . . . . . . 12

3.1.4 Segment Filter . . . . . . . . . . . . . . . . . . . . . . . . . 12

3.2 Common Class of List Operations . . . . . . . . . . . . . . . . . . . 13

**4 Index Compression Strategies** . . . . . . . . . . . . . . . . . . . . . . . **14**

**5 Online Index Compression** . . . . . . . . . . . . . . . . . . . . . . . . **19**

5.1 Challenges for String Similarity Join . . . . . . . . . . . . . . . . . . 19

5.2 Extending Proposed Approaches . . . . . . . . . . . . . . . . . . . . 21

5.3 Benefit Estimation Model based Algorithm . . . . . . . . . . . . . . . 24

**6 Discussion** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **28**

6.1 Extension for Offline Approaches . . . . . . . . . . . . . . . . . . . . 28

6.2 Online Approaches . . . . . . . . . . . . . . . . . . . . . . . . . . . 28

6.2.1 Cache-aware design . . . 29
6.2.2 SIMD-aware design . . . 29

**7 Evaluation . . . 31**

7.1 Experiment Setup . . . 31
7.2 Effect of Compression Techniques . . . 32
7.3 End-to-end Query Time . . . 35
7.4 Scalability . . . 38
7.5 Case Study . . . 40

**8 Related Work . . . 41**

8.0.1 String Similarity Search and Join . . . 41
8.0.2 Data Compression . . . 41

**9 Conclusion . . . 43**

**References . . . 44**

LIST OF FIGURES

2.1 Two-layer compression scheme . . . . . . . . . . 7

2.2 Fix-length Compression in MILC . . . . . . . . . . 8

4.1 Variable-length Compression (CSS) . . . . . . . . . . 15

5.1 Examples for online fix-length compression (a) and online variable-length compression (b) . . . . . . . . . . 22

7.1 Index Time for Similarity Search . . . . . . . . . . 33

7.2 Comparison of Execution Time: Similarity Search . . . . . . . . . . 36

7.3 Comparison of Execution Time: Similarity Join . . . . . . . . . . 37

7.4 Scalability: Index Size . . . . . . . . . . 38

7.5 Scalability: Execution Time . . . . . . . . . . 39

LIST OF TABLES

7.1 Statistics of Datasets . . . . . 31

7.2 Index Size for Compression Schemes: Similarity Search (MB) . . . . . 33

7.3 Index Size for Compression Schemes: Similarity Join (MB) . . . . . 34

7.4 Index Size: Amazon Review . . . . . 40

## ACKNOWLEDGMENTS

This thesis is based on the existing work I authored [47] published in the 38th IEEE International Conference on Data Engineering (ICDE 2022). I would like to thank Dr. Jin Wang, Dr. Chunbin Lin, and Professor Carlo Zaniolo for their collaboration and contributions to the project: they have provided insightful input and helped to refine the ideas. Furthermore, Professor Carlo Zaniolo has provided the funding that made the publication of this work possible.

Furthermore, I would like to express my deep appreciation once again to Dr. Jin Wang for providing the initial support in the research process, Professor Carlo Zaniolo for his constant guidance in numerous discussions throughout various projects, and Dr. Chunbin Lin, Dr. Zhiyi Zhang, and Professor Lixia Zhang for their invaluable assistance in the research endeavor. Their expertise and commitment have been instrumental in my research journey, and I am deeply grateful for their help.

Finally, I would like to express my sincere gratitude to Professor Todd Millstein and Professor George Varghese for taking the time out of their busy schedules to serve as my committee members.

# CHAPTER 1

# Introduction

String similarity searches and string similarity joins represent fundamental operations in many real world applications, including information retrieval [6], data mining [33], entity extraction [42], near duplicate detection [46], data cleaning [11] and integration [44]. Given a query string and a data collection, similarity search aims at finding all strings from the collection that are similar to the pattern specified by the query; given two string collections, the similarity join operator aims at finding all similar string pairs in the two collections. Two strings are similar to each other if their similarity exceeds a predefined threshold under particular metrics, such as the token-based metrics of Jaccard, Cosine, and Dice, and the character-based metrics of Edit Distance.

The field of data management has witnessed a substantial body of research aimed at addressing the challenge of string similarity search and join [32]. Most of the existing solutions employ a filter-and-verification framework [11, 7, 46, 18, 28, 37]. This framework consists of a filtering phase and a verification phase. During the filtering phase, various techniques are employed to generate signatures from the input strings and to construct an inverted index based on these signatures. The inverted index is used to narrow down the set of candidate strings by pruning out those that are dissimilar. Finally, in the verification phase, the similarity values of the remaining candidates are computed to determine the final answers.

**Motivation for Compressed Indexes** Past studies in the realm of string similarity search and join have focused on the development of effective filtering methods to achieve

high pruning power, such as the count filter [21], prefix filter [11], position filter [46] and segment filter [28, 18]. However, a significant limitation of these approaches is the memory overhead imposed by the inverted indexes. As the size of the inverted index grows, the list operations (e.g., intersection, union, insertion, existence checking) over the posting lists become the performance bottleneck in the context of string similarity search and join. When the inverted lists are too large to fit in memory, disk-based or distributed algorithms must be employed, which incur significant additional costs associated with disk or network I/O.

*To address the above challenges, in this thesis, we seek to speed up the performance over inverted index in the presence of large datasets by implementing two main strategies: (i) utilizing online compression algorithms to significantly reduce the space overhead of the inverted index through compression and (ii) performing list operations directly on the compressed index without the need for decompression.*

**Limitation of Existing Compression Schemes** Numerous compression techniques have been introduced in the domains of Information Retrieval and Databases. However, they are not well-suited for string similarity search and join operations. For instance, PforDelta [51] and its variations [49, 48], which represent the most widely-used methods for compressing inverted lists, require decompression to access elements from the compressed data. On the other hand, while MILC [41] addresses the above limitations by supporting query processing on compressed data, it is still not applicable for *online* index construction, which is crucial in similarity join algorithms. Similar limitations hold for compression methods used in file systems and databases, such as Bzip [1], Zlib [3], and LZO [2], which are highly efficient and can significantly reduce I/O costs. Bitmap compression techniques utilized in databases include WAH [25], PLWAH [17], CONCISE [14], and Roaring [10]. However, these techniques cannot be used for online index compression as they cannot handle incremental updates efficiently.

For efficient string similarity search and join operations using compressed inverted indexes, a compression method is needed that satisfies the following requirements: (i) it mus

allow direct querying on compressed data without the need for decompression, (ii)it mus facilitate online construction and efficient updating of the compressed index, and (iii) it must provide lossless compression to ensure that accurate results are produced. Currently, however, there is no compression method that satisfies all three requirements, and critical research is needed to fill this gap.

**Our Solution** In this this thesis, we seek to fill this crucial gap by proposing a general compression framework, named **C**ompressed **S**tring **S**imilarity (CSS), which supports a broad spectrum of frameworks for string similarity search and join. We begin by summarizing existing filtering techniques and defining a set of list operations required for these techniques. We then examine the state-of-the-art compression method MILC [41], which utilizes a two-layer storage structure for compressed data, allowing for direct list operations without decompression overhead. Building on such structure, we further design a more flexible scheme CSS, which not only gives a higher compression ratio, but also supports *online* inverted-list compression during string similarity joins. Additionally, CSS can reduce computational redundancies, leading to query performance that is comparable to, or even faster than, those obtained using uncompressed lists.

We start from the similarity search problem and utilize offline compression schemes to partition and compress the lists into compressed blocks. However, these schemes can not be directly applied to similarity joins, where inverted indexes are generated on-the-fly. Therefore, we propose a buffer-based scheme to extend the offline compression schemes to online scenarios. The fundamental concept involves dividing a list into compressed and uncompressed segments, where the compression is performed based on the evaluation of predefined predicates, with the corresponding benefit derived from such evaluation. In order to balance the computation overhead and memory consumption effectively, a benefit model is introduced to analyze the sequence of elements in the list, thus guiding the formation of compressed blocks. As a result of the benefit model, this adaptive compression scheme demonstrates

remarkable compression capability with minimal computational overhead, thus obviatin the need for hyperparameter tuning. Due to their efficient and lightweight structure, these compression techniques can reduce memory overhead while producing query processing performance that is comparable to that obtained by directly querying uncompressed lists.

To summarize, the main contributions we made with this work are as follows:

- To the best of our knowledge, CSS is the first work that systematically analyzes lossless compression techniques seeking to reduce the memory footprints and the runtimes spent in string similarity search and join applications.
- We conducted an extensive study of existing frameworks for string similarity search and string similarity join. Furthermore, based on our study, we summarized a suite of necessary operations over the compressed inverted index lists required by these frameworks that CSS needs to support.
- We provided several compression schemes tailored to string similarity search and join. Specifically, we proposed the first solution for inverted index compression in an online manner to support similarity join and develop three efficient online compression schemes based on a benefits model.
- We conducted an extensive set of experiments on several real-world datasets. The outcomes demonstrate that CSS attains comparable, or even better, query performances than its uncompressed counterparts, while achieving a substantial reduction in memory usage.

The rest of the thesis is organized as follows: Chapter 2 introduces necessary background knowledge. Chapter 3 summarizes existing filtering techniques for string similarity search, and string similarity join to identify the common list operations among them. Chapter 4 and Chapter 5 propose the new compression scheme for offline and online index construction, respectively. Chapter 6 addresses important considerations for the broader applications of CSS. Chapter 7 reports the results of the evaluation. Chapter 8 surveys the related work. Finally, Chapter 9 concludes the whole thesis.

# CHAPTER 2

# Preliminary

## 2.1 String Similarity Search and Join

We first formally define the problem of string similarity search and string similarity join. Let $\mathcal{R}$ and $\mathcal{S}$ be two string collections, where $r \in \mathcal{R}$ and $s \in \mathcal{S}$ represent individual strings. In order to use filter-and-verification techniques, it is necessary to convert the strings into signatures, such as $q$-grams, and we use $SIM(s, r)$ to represent the similarity between strings $r$ and $s$. The $i$-th character in $r$ or $s$ is denoted as $r[i]$ or $s[i]$, respectively.

The formal definitions of string similarity search and join are given in Definition 1 and Definition 2, respectively [1].

For the purpose of this discussion, we will focus on similarity self-join. However, the techniques presented can be easily extended to the case of a join between $\mathcal{R}$ and $\mathcal{S}$.

**Definition 1** (String similarity search (SSS))**.** *Given a query $r$, a string collection $\mathcal{S}$ and a threshold $\tau$,a string similarity search operation aims at finding all strings $s \in \mathcal{S}$ satisfying $SIM(r, s) \geq \tau$.*

**Definition 2** (String similarity join (SSJ))**.** *Given a string collection $\mathcal{S}$ and a threshold $\tau$, a string similarity join operation aims at finding all strings $r, s \in \mathcal{S}$ where $s \neq r$ and $SIM(r, s) \geq \tau$.*

---

[1] For edit distance, the condition of similar strings is that the value is no larger than the given threshold $\tau$.

The key distinction between similarity search and join are discussed next. In the case of similarity search, the indexes, i.e., the inverted lists, are constructed in the offline step without having prior knowledge of the threshold. Once the query string and threshold are given, results are obtained by traversing the inverted lists. On the other hand, in the case of similarity join, the index is constructed during the online step as part of the join process, as it is dependent on the value of the threshold, which is only known at run-time. As a result, the time required for index construction is considered part of the join time in similarity joins. To accommodate these requirements, we need to apply different compression techniques for offline and online index construction.

## 2.2 Inverted List Compression

To reduce memory consumption, we focus on compressing the size of inverted lists, which represents the bottleneck in memory usage. Inverted lists in SSS and SSJ store the record IDs (elements) of all records that contain a specific signature in their posting list. These elements are represented by unique, sorted positive integers. Our goal is to store these lists using as few bits as possible while still allowing for efficient query processing on the compressed list. To better understand the compression mechanisms, we will first introduce some fundamental concepts. The "compression ratio" used in this work is defined as $r = \frac{U}{C}$, where $U$ represents the size of the original uncompressed data and $C$ represents the size of the compressed data. For instance, if a list consists of 10 integer elements, each using 32 bits, then $U = 10 \times 32 = 320$. If a compression method compresses the list into 200 bits, then the compression ratio of this method would be $r = \frac{U}{C} = 1.6$. A higher compression ratio means more memory space saved.

In this paper, we will employ the two-layer compression scheme proposed by MILC [41] as the cornerstone of our proposed framework. The core benefit of this two-layer compression scheme is that it supports efficient random access directly to compressed data without de-

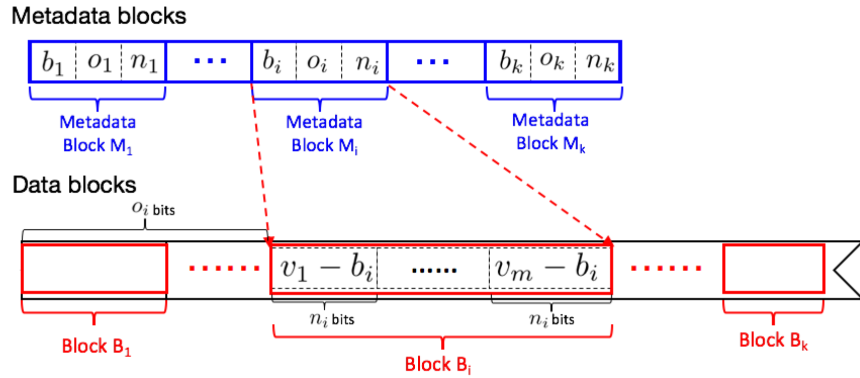



Figure 2.1: Two-layer compression scheme

compression, which is a crucial requirement for the set of operations outlined in Section3.2. A visual representation of the details can be seen in Figure 2.1. More precisely:

- Metadata Layer: This layer contains a list of metadata blocks, each of which corresponds to a compressed data block in the data layer. Each metadata block $M = (b, o, n)$ holds three elements: (i) the base value $b$, which represents the first element in the corresponding data block; (ii) the offset bit $o$, which is the sum of the bits used by the previous data blocks; and (iii) the number of bits $n$ used by each element in the corresponding data block. It is important to note that each $M$ requires 69 bits to store, with 32 bits for $b$, 32 bits for $o$, and 5 bits for $n$ (as the maximum number of bits used by an element in a data block is 32, which can be expressed in 5 bits).
- Data Layer: This layer contains a list of compressed data blocks $B$. The compressed data blocks are organized into 32-bit words. Instead of storing the original value $v$, we store $v - b$, which represents the difference between $v$ and the base value $b$. Each difference is stored using $n$ bits within the block.

Based on the above settings, MILC then employs a fixed-length partitioning and compression method. More specifically, MILC takes a sorted list $L$ and the number of elements $m$ per block as inputs and partitions $L$ into $\lceil \frac{|L|}{m} \rceil$ blocks $B_1, ..., B_k$. If $v_0, v_1, ..., v_{m-1}$ is the list of elements stored in block $B_i$, then $v_0$ is designated as the base element and stored in

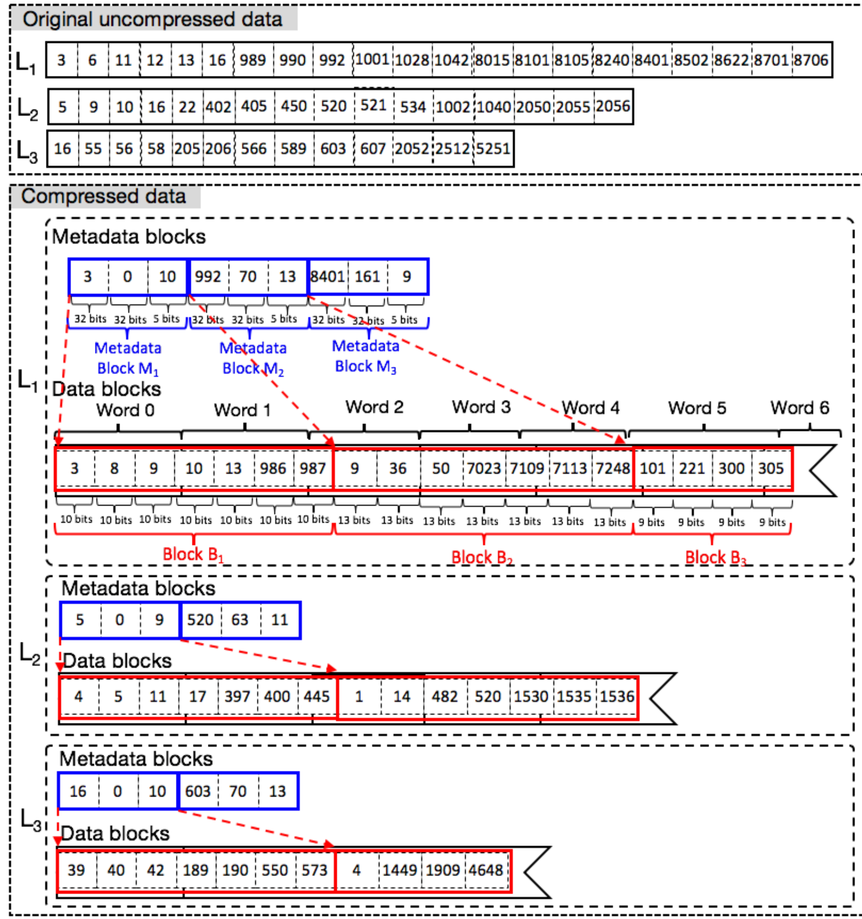



Figure 2.2: Fix-length Compression in MILC

the metadata block $M_i$, i.e., $b = v_0$. While $v_0$ is stored in the metadata block, the actual values stored in $B$ are the offsets $(v_1 - b), (v_2 - b), ..., (v_{m-1} - b)$. Now, each offset value takes $n = \lceil \log(v_{m-1} - b + 1) \rceil$ bits in the data block $B$. To enable random access to the compressed list, $n$ is also stored in metadata block $M_i$. Let $o_{i-1}$ be the offset bit stored in the previous metadata block $M_{i-1}$, then $o_i = o_{i-1} + n(m-1)$ is stored in the current metadata block $M_i$.

# CHAPTER 3

# A Unified Compression Framework

In this chapter, we discuss the requirements that a unified compression scheme must satisfy to support string similarity search and string similarity joins efficiently. We first summarize the existing filtering techniques in Section 3.1 and then, in Section 3.2, we discuss the operations required for their efficient support on compressed inverted lists.

## 3.1 Existing Filtering Techniques

Most existing solutions employ a filter-and-verification framework for string similarity search and similarity search join. More specifically, in the filter step, these solutions employ various filtering techniques to reduce the number of candidates, which will be discussed later. For clarity, here, we will use the Jaccard similarity metric to illustrate these techniques. However, our compression methods are not restricted to specific similarity metrics or filtering techniques.

### 3.1.1 Count Filters

The core idea of Count Filters [21] is that two strings should be considered similar only when they share a sufficient number of signatures. The filtering problem can then be transformed into the problem of solving the $T$-Occurrence [21], where the value of $T$ is determined based on the chosen similarity metric. Given two strings $r$ and $s$, with signature sets $Sig(r)$ and $Sig(s)$, respectively, $r$ is considered similar to $s$ (i.e. $\text{JAC}(r, s) \geq \tau$) if:

$$Sig(r) \cap Sig(s) \geq \frac{\tau(|Sig(r)| + |Sig(s)|)}{1+\tau} \tag{3.1}$$

To reduce the cost of traversing the lists in the Count Filter, several list merging algorithms were proposed in [26]. The most widely used of these is MergeSkip, which uses binary search to avoid traversing the entire list.

The main idea behind MergeSkip is to skip lists that cannot contain any element in the answer set of the query with threshold $T$. Thus, a heap is kept for the frontiers of these lists, and all elements are popped which have the same value as the top element $e$ in the heap. If there are $T$ such elements, $e$ is added to the answer set; otherwise, $e$ cannot be an answer. Additionally, MergeSkip further pops some additional lists up to a threshold related to $T$ and performs binary searches over all popped lists to locate its smallest element $e_{min}$ such that $e_{min} \geq e'$ and push $e_{min}$ into the heap, where $e'$ is the current top of the heap. In this way, MergeSkip can skip over a lot of elements that are smaller than the $e_{min}$ values in these lists, since these elements cannot produce answers to the query.

### 3.1.2 Prefix Filters

A Prefix Filter [11] improves the filtering power and reduces the filtering cost by focusing on the prefix of all strings. It starts by establishing a global order $\mathcal{O}$ for all tokens in the dataset. Then, for a string $s$, the tokens are sorted according to $\mathcal{O}$ and the $\tau$-prefix of $s$ is represented as $\mathcal{P}^s_\tau$. In the case of Jaccard similarity, dissimilar strings can be filtered out using Lemma 1.

**Lemma 1** (Prefix Filter [11])**.** *Given two strings $s$, $r$ and a threshold $\tau$, the length of $\mathcal{P}^r_\tau$ ($\mathcal{P}^s_\tau$) is $\lfloor (1-\tau)|r| + 1 \rfloor$ ($\lfloor (1-\tau)|s| + 1 \rfloor$). If $\mathcal{P}^r_\tau \cap \mathcal{P}^s_\tau = \emptyset$, then we have* $\mathrm{JAC}(r, s) < \tau$.

The process of using the Prefix Filter to perform string similarity join is shown in Algorithm 1. Thus, the algorithm first initializes the result set and the inverted lists set (line: 2), and then, for each string in the collection, it generates the corresponding prefix according

to Lemma 1 (line: 4). The algorithm then traverses the inverted lists that correspond to signatures in the prefix and retrieves the potential matches (line: 6). For all the potential matches, a verification is performed to identify similar pairs (line: 8). After a string $s$ is processed, it is appended to the inverted lists associated with its prefix for further comparisons with subsequent strings (line: 11). Finally, the results are returned after all strings have been processed (line: 12). Note that in this process, the inverted index is constructed in an online manner. Furthermore, a similar approach can be used to perform other string similarity joins, including those that use Position Filters and Segment Filters, which are discussed next.

**Algorithm 1**: Index Construction($\mathcal{S}$, $\tau$)

```
Input: S: The string collection; τ: The similarity threshold.
Output: A: set of pairs ⟨r, s⟩ s.t. r, s ∈ S, r ≠ s, JAC(r, s) ≥ τ
1  begin
2      Initialize the result set A and the inverted lists L as ∅;
3      for s ∈ S do
4          foreach signature e ∈ P_τ^s do
5              foreach r in L[e] do
6                  if r is not visited and pass the length filter then
7                      if JAC(s, r) ≥ τ then
8                          Add ⟨r, s⟩ into A;
9
10
11             Append s into inverted list L[e];
12     return A;
13 end
```

### 3.1.3 Position Filter

The Position Filter [46] builds on top of the Prefix Filter. The core idea is to check the difference in position between the matched signatures after finding them in the prefix. Given two strings $r$ and $s$ and a signature $e$ that exists in the prefix of both $r$ and $s$, with the position of $e$ in $\mathcal{P}_\tau^r$ and $\mathcal{P}_\tau^s$ represented as $P_e[r]$ and $P_e[s]$, respectively, if $\text{JAC}(r,s) \geq \tau$, then for any $e \in \mathcal{P}_\tau^r$ ($\mathcal{P}_\tau^s$), the following constraint must hold: $|r| - P_e[r] + 1 \geq \frac{\tau(|Sig(r)|+|Sig(s)|)}{1+\tau}$.

The filtering power of the Position Filter can be further enhanced by considering the suffix of the strings. This filter can be integrated into Algorithm 1 by adding an additional check before verifying the results.

### 3.1.4 Segment Filter

In contrast to the aforementioned filtering techniques, the Segment Filter splits all strings into non-overlapping partitions, and two strings are considered similar only if they have at least one common partition. The Segment Filter was first introduced for edit distance [28] and later extended to support set-based similarity metrics, such as Jaccard, in [18].

For Jaccard similarity, the following condition applies. Given two strings $r$ and $s$ ($|r| < |s|$) split into $k$ non-overlapping partitions with the same partition scheme, where $k = \lceil \frac{1-\tau}{\tau} * |r| \rceil + 1$, $\text{JAC}(r,s) \geq \tau$ if the $l$-th partition of $s$ and $r$ is the same for some $l$ in the range of $[i,k]$.

In the above filtering techniques, the Count Filter can be applied to both the similarity search and similarity join problems, whereas other filtering techniques are designed for sting similarity joins since they require a threshold $\tau$ to be specified as part of the input.

## 3.2 Common Class of List Operations

Based on the above analysis, we conclude that, in order to support common filtering techniques, we should provide efficient implementations for the following operations on compressed inverted lists:

- Verification: Check whether an element exists in one list.
- Intersection: Find common elements of multiple lists.
- Union: Find the union of all elements of multiple lists.
- Insert: Append one or many new elements to the end of a list.

Among the filtering methods, the Count Filter requires Verification and Intersection operations, while the Prefix Filter, Position Filter, and Segment Filter all require Union operations. For string similarity joins, all filtering methods require the insertion of elements to build the inverted lists incrementally. Additionally, the Prefix and Position Filters require additional fields to store the position of signatures within each string. As outlined in Section 2.2, current state-of-the-art methods face significant overhead in query processing and are unable to build compressed indexes incrementally. In the following section, we will present new compression schemes that can efficiently support these operations.

# CHAPTER 4

# Index Compression Strategies

In this chapter, we study the inverted list compression techniques under offline setting. As introduced in Section 2.2, the state-of-the-art approach MILC employees a fix-length partition scheme. Since MILC partitions data into equal-size blocks, it will achieve a high compression ratio if data is (almost) evenly distributed. However, that is not always the case in real-life applications, and in situations where there is skewness in the data distribution, MILC will result in a significant waste of space.

**Example 1.** *Given the list $L_1$ with 21 elements shown in Figure 2.2, suppose MILC partitions $m = 8$ elements into one block, then MILC evenly splits $L_1$ into $\lceil \frac{21}{8} \rceil = 3$ data blocks $B_1$, $B_2$ and $B_3$. Thus three corresponding metadata blocks $M_1$, $M_2$, and $M_3$ are created. Each metadata block occupies 69 bits, as we introduced before. The start values of $B_1$, $B_2$ and $B_3$ are $M_1.b_1 = 3$, $M_2.b_2 = 992$ and $M_3.b_3 = 8401$, respectively. Since MILC stores the offset of original values to the start value in each block, the actual values stored in blocks can be decided accordingly. Take $B_1$ as an example, its values can be calculated as $\{3, 8, 9, 10, 13, 986, 987\}$. Therefore, the number of bits needed to represent the data in $B_1$, $B_2$ and $B_3$ are $B_1.n_1 = \lceil \log(987 + 1) \rceil =$ 10, $B_2.n_2 = \lceil \log(7248 + 1) \rceil =$ 13 and $B_3.n_3 = \lceil \log(305 + 1) \rceil =$ 9. The total number of bits of $B_1$, $B_2$ and $B_3$ are $10 \times 7 = 70$, $13 \times 7 = 91$, and $9 \times 4 = 36$, while $M_2.o_2 = 70$ [1] and $M_3.o_3 = 70 + 91 = 161$. In total, MILC uses $69 \times 3 + 70 + 91 + 36 = 404$ bits to represent the whole list while the original size is $32 \times 21 = 672$ bits. The compression ratio of MILC in this example is $\frac{672}{404} = 1.66$.*

[1] *$M_1.o_1$ is always 0.*

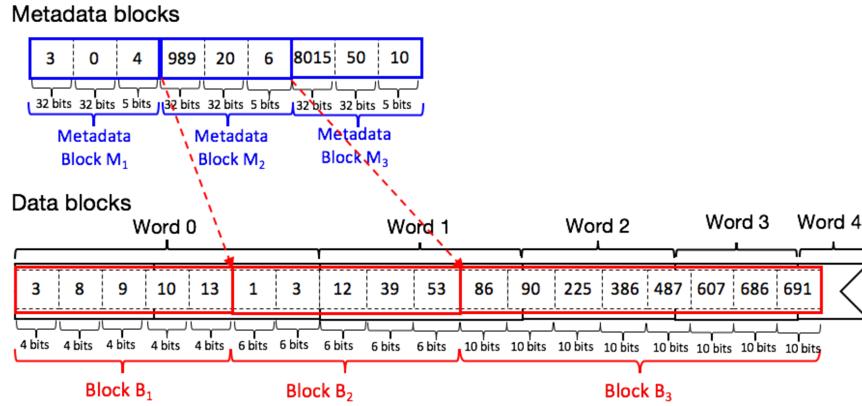



Figure 4.1: Variable-length Compression (CSS)

In Example 1, it is clear that MILC produces sub-optimal partitions due to data skewness. In fact, $B_1$ will need 10 bits to store each element due to the existence of $\{986, 987\}$, which are the direct result of the fix-length partition scheme of MILC.

To address the limitations of MILC and provide a more robust compression scheme, we present a novel variable-length compression scheme, referred to as CSS. This scheme accommodates the capacity to allocate varying numbers of bits per element to different blocks, thereby providing greater flexibility and robustness. To this end, we employ a dynamic programming approach to identify the optimal partitioning positions to minimize the total number of bits. By properly allocating elements into different blocks, we can significantly reduce the space overhead. For example, as demonstrated in Figure 2.2 and Example 1, when the elements 989 and 990 are placed into block $B_1$, each element therein requires 10 bits to store. On the other hand, if these elements were to be inserted into block $B_2$, the number of bits required for each element in $B_1$ would be reduced to 4, resulting in a substantial decrease in terms of space overhead.

**Details of CSS.**

To minimize the overall space overhead, we need to ensure that the memory space saved through the creation of a new data block is greater than the overhead incurred by the

associated metadata block, which has a fixed size of 69 bits. To this end, we employ a dynamic programming algorithm as the solution. The goal is to find a set of positions $P$ in $L$ that decides the partitions, using the benefits we gain when CSS creates a new data block as the decision criteria.

Let $G[x, y]$ denote the space we can save once the sequence from $x$-th to $y$-th elements on $L$ is regarded as a block. If the number of bits an element in this block occupies is $b$, then the number of saved bits $G[x, y]$ can be calculated as $(y - x) * (32 - b) + 32 - 69$. If $OPT[i]$ denotes the maximum saved space in the sub-list $L[1...i]$, then we can calculate the value of $OPT[i]$ using the following Equation (4.1):

$$OPT[i] = \max_{0<j<i} G[j, i] + OPT[j] \tag{4.1}$$

Algorithm 2 shows details of the Variable-length approach. The algorithm first initializes the value of the benefit, which is negative with just one element in a block (line: 2) due to its corresponding metadata block. Then it traverses each position in $L$ and initializes the value of the maximum benefit in the current sub-list (line: 4) and the partition positions (line: 5). Then in the inner loop, it will calculate the maximum benefits as well as the corresponding partition positions (line: 6-9). Finally, the set of partition positions $P$ is returned, resulting in the maximum space-saving, i.e., the minimum number of bits used in total. The two-level block structure can be constructed according to the information contained in $P$.

The time complexity of Algorithm 2 is $\mathcal{O}(n^2)$, where $n$ is the length of $L$. However, as this process occurs during the index construction phase in an offline setting, it does not impact the performance of the string similarity search process. As shown by the experiments reported in Figure 7.1, the overhead of this process is also reasonable. Additionally, we can also add some constraints, e.g., the maximum size of a data block, to reduce the cost of this process when necessary.

**Example 2.** *Recall the inverted list in Figure 2.2, suppose* CSS *will split the list into three*

**Algorithm 2**: Variable-length Partition ($L$, $n$)

**Input**: $L$: The inverted list; $n$: Cardinality of $L$; $G$: Pre-caculated benefit matrix

**Output**: $P$: set of positions to split $L$

```
1  begin
2      Initial OPT[0] = 32 − 69 = −37;
3      for i = 1 to n do
4          OPT[i] = G[0, i];
5          P[i] = i;
6          for j = 0 to i − 1 do
7              if OPT[j] + G[j+1,i] > OPT[i] then
8                  OPT[i] = OPT[j] + G[j+1,i];
9                  P[i] = j;
10
11     return P;
12 end
```

*blocks with starting values* 3, 989, *and* 8015 *respectively, as shown in Figure 4.1. Here it creates three corresponding metadata blocks* $M_1$, $M_2$, *and* $M_3$. *Each metadata block occupies 69 bits, as introduced before. The values stored in each block are computed in a similar way. For example, we have* $B_3$ *as* $\{86, 90, 225, 386, 487, 607, 686, 691\}$. *Therefore, the number of bits needed to represent the data in* $B_1$, $B_2$, *and* $B_3$ *are* $B_1.n_1 = \lceil\log(13 + 1)\rceil = 4$, $B_2.n_2 = \lceil\log(53 + 1)\rceil = 6$ *and* $B_3.n_3 = \lceil\log(691 + 1)\rceil = 10$. *The total number of bits for* $B_1$, $B_2$ *and* $B_3$ *are* 20, 30, *and* 80, *respectively. In total, CSS uses* $69 \times 3 + 20 + 30 + 80 = 337$ *bits to represent the whole list while the original size is* 672 *bits. Meanwhile, we can see that the compression ratio of CSS is* $\frac{672}{337} = 1.99$.

**Binary Search over Compressed List** Another issue to resolve is how to quickly traverse the compressed list. To this end, we employ a binary search algorithm described as follows:

1. *Metadata lookup.* Given a search key $e$, it first performs the binary search over the metadata block by comparing with the base value $M.b$ to find the data block $B_i$ such that $M_i.b_i \leq e$ and $M_{i+1}.b_{i+1} > e$.

2. *Data lookup.* Then it uses $e - M_i.b_i$ as a new search key in data block $B_i$ to perform the binary search starting from address $M_i.o_i$, where the $t$-th element in $B_i$ is located at the address of $M_i.o_i + M_i.n_i \times (t - 1)$.

An important observation to be made here is that the binary search is performed directly over the compressed data without decompression, thus significantly reducing the query processing overhead.

**Example 3.** *Recall the list compressed in Figure 2.2 and a search key* 8015*. It first does metadata lookup to find the block* $B_2$ *since* $M_2.b_2 = 992 < 8015$ *and* $M_3.b_3 = 8401 > 8015$*. Then it uses the new key* $8015 - 992 = 7023$ *to do the binary search in Block* $B_2$ *starting from* 70 *bit, and the* 4*-th value is at address* $70 + 13 \times (4 - 1) = 109$ *bit, and the size is* 13 *bit. So we get the value of* 7023*, which matches the new search key.*

With the ability to perform the binary search directly over compressed data, the MergeSkip algorithm can be efficiently supported on the compressed inverted lists.

# CHAPTER 5

# Online Index Compression

In this chapter, we investigate compression techniques for string similarity joins, which require compressed indexes to be constructed on-the-fly during query processing. We first describe the new challenges that need to be addressed in the online scenario in Section 5.1. Next, we discuss how to extend the two offline compression methods (i.e., MILC and CSS) proposed in Section 4 and propose their online versions in Section 5.2. Finally, in Section 5.3, we propose a cost model to evaluate the trade-offs when doing online compression and devise a new adaptive compression approach accordingly. Since all of the approaches discussed in this section are based on our CSS framework, we will omit the prefix CSS when referring to these approaches(e.g., we use the term Adapt when referring to $\mathsf{CSS}_{Adapt}$).

## 5.1 Challenges for String Similarity Join

While the compression techniques outlined in Section 4 are effective for addressing the problem of string similarity search, they do not support the applications of string similarity join well. The reason for this limitation is that these techniques require the complete list as an input, meaning that the compressed index is constructed in the offline stage. However, as demonstrated in Algorithm 1, string similarity joins necessitate constructing the index in an online fashion. Consequently, two new challenges must be addressed:

- **Compression decision making**. The compression techniques employed must determine whether to form a new data block as soon as an item arrives. Since the entire

list is still unavailable when making the decision, this will will inevitably result in a reduction of the expected compression ratio.

- **Supporting incremental update**. The compressed index must be updated incrementally as more records in the dataset are accessed. Given that the index construction time is part of the overall time required for the string similarity join, it is necessary to avoid reconstructing the compressed index during the entire process.

Nevertheless, the online compression scheme can be developed by extending the the fix-length and variable-length approach we have proposed. To reach this goal, we should enable the incremental maintenance on the two-level block structure. However, it is necessary to identify the number of bits used to encode elements in each block before creating the metadata blocks. Although this value can be easily determined by scanning the entire list, it is not known ahead of time in online settings. A simple solution involves adjusting the metadata and data blocks after each element arrives. However, this approach can be prohibitively expensive as each data block $i$ must be reorganized once the value of $b_i$ changes.

An additional detail that must be addressed is the need to store both ID and position information in the inverted list for certain filtering techniques, such as the Prefix Filter and the Position Filter. This issue can be resolved by storing these pieces of information separately in two lists. The list of IDs can be stored using the compression techniques described later in this thesis. However, since the list is unsorted, these techniques cannot be applied to store the corresponding position information. Instead, a straightforward strategy employing the same number of bits as the largest element in the list of positions will be utilized.

## 5.2 Extending Proposed Approaches

To address the challenges created by the online settings, we next propose a lazy updated block structure. Specifically, the inverted list is divided into two regions: the compressed region and the uncompressed region. The compressed region is identical to that of the offline setting, while the uncompressed region serves as a buffer for incoming elements. When a new element arrives, we first check whether the uncompressed region is full. If so, a new data block is created for all elements in the uncompressed region and moved to the compressed region. Otherwise, the new element is simply added to the buffer.

During the processing of a string similarity join, elements in each list arrive in ascending order, and the uncompressed region always contains elements with larger IDs. As a result, random access to the list can still be supported by visiting the two regions separately.

**Online Extension for MILC (Fix)** It is rather straightforward to apply this idea to the fix-length approach and extend it to the online setting. Since the size of all data blocks is fixed at $m$, we can also set the cardinality of the uncompressed region to $m$. When a new block needs to be created, the first element in the uncompressed region is regarded as the base. Then offsets are calculated accordingly, and $b_i$ is assigned the value of the maximum delta among them. Finally, the blocks are appended to the compressed region, and the uncompressed one is clear.

**Online variable-length compression (Vari).** The variable-length approach can be adapted to the online setting in a similar manner to the fix-length approach. However, a new challenge arises in determining an appropriate value for the maximum cardinality of the uncompressed region. If this value is too small, the optimal solution may be missed, while setting it too high results in significant overhead in both memory usage and the execution of the dynamic programming algorithm. To identify a proper uncompressed region size, we perform a detailed analysis of the number of saved bits in the variable-length approach. The findings are summarized by Theorem 1.

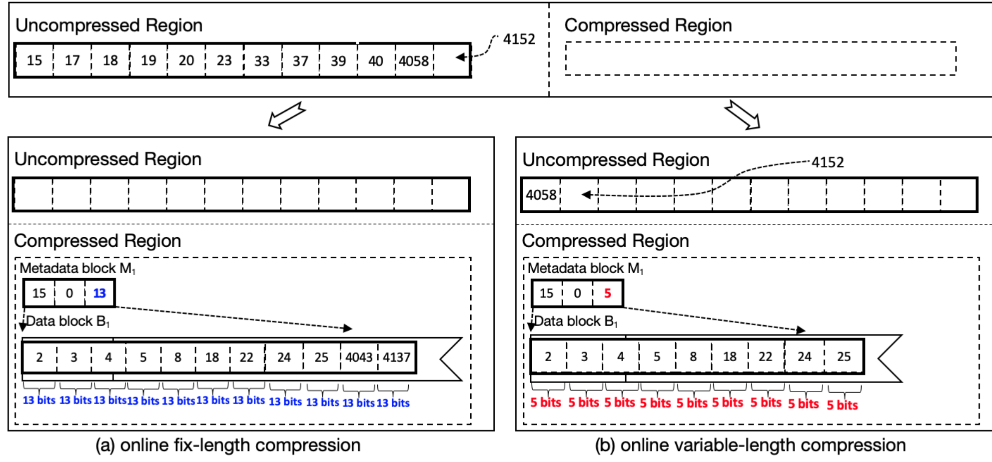



Figure 5.1: Examples for online fix-length compression (a) and online variable-length compression (b)

**Theorem 1.** *Let $|M|$ be the total number of bits needed for a metadata block, then the lower bound and upper bound of the size of a data block with the variable-length compression (CSS) is $\lceil \frac{|M|}{32} \rceil$ and $2 \times |M|$, respectively.*

*Proof.* We will prove that the lower bound is $\lceil \frac{|M|}{32} \rceil$ by contradiction. Suppose that there is an uncompressed region $X$ with 2 elements, where $G[0,1] \geq 0$, and let $X_i$ be the $i$-th element of $X$. Then we have:

$$32 \times 2 - \lceil \log_2(X_1 - X_0 + 1) \rceil \times 2 - 69 >= 0 \tag{5.1}$$

Since each element must differ by at least 1, we know that $\lceil \log_2(X_1 - X_0 + 1) \rceil > 0$. Thus, it is impossible for $32 - \lceil \log_2(X_1 - X_0 + 1) \rceil \geq 34.5$ to be true. This implies that the lower bound holds.

Next, we prove the upper bound is $2 \times |M|$. Assuming there is an uncompressed region $X$ with size $m > 138$, where the i-th element in $X$ is denoted by $X_i$.

If we were to compress the entire uncompressed region, we would have the following result:

$$G[0, m-1] = 32 \times m - \lceil log_2(X_{m-1} - X_0 + 1) \rceil \times m + 69) \tag{5.2}$$

Next, we partition this region into two blocks, with the dividing point being the middle element $X_k$. The total gain from this partition would be:

$$G[0, k-1] + G[k, m-1] = 32 \times m - [\lceil log_2(X_{m-1} - X_k + 1) \rceil \times (m - k + 1) + \lceil log_2(X_{k-1} - X_0 + 1) \rceil \times (k) + 138] \quad (5.3)$$

We then introduce a new gain, $G'$, which is always smaller than $G[0, k-1] + G[k, m-1]$:

$$G' = 32 \times m - [\lceil log_2(X_{m-1} - X_k + 1) \rceil \times (m/2) + \lceil log_2(X_{k-1} - X_0 + 1) \rceil \times (m/2) + 138] \quad (5.4)$$

To show that the upper bound holds, we must prove that $G'$ is always greater than $G[0, m-1]$. By simplifying the equation, we get:

$$m[2 \times \lceil log_2(X_{m-1} - X_0 + 1) \rceil - \lceil log_2(X_{m-1} - X_k + 1) \rceil - \lceil log_2(X_{k-1} - X_0 + 1) \rceil] > 138 \quad (5.5)$$

Since we have already assumed that $m > 138$, all that needs to be proven is that:

$$2 \times \lceil log_2(X_{m-1} - X_0 + 1) \rceil - \lceil log_2(X_{m-1} - X_k + 1) \rceil - \lceil log_2(X_{k-1} - X_0 + 1) \rceil >= 1 \quad (5.6)$$

This equation can be simplified further to $X_{m-1} - X_0 + 1 < X_{m-1} - X_k + 1 + X_{k-1} - X_0 + 1$, which leads to $X_k - X_{k-1} < 1$. However, this is a contradiction, so the upper bound of $2 \times |M|$ holds.

Our proof is thus completed. □

On the basis of the above findings, we establish the maximum number of bits occupied by the uncompressed region as: $69 * 2 = 138$. When a new element arrives, and the uncompressed region is full, a dynamic programming process similar to Algorithm 2 is applied to all elements in the region. Unlike the offline algorithm, only one data block is created in the compressed index each time. The remaining elements remain in the uncompressed region, awaiting the next time the buffer reaches full capacity. As a result, the time complexity of the dynamic programming process for each element is $\mathcal{O}(|M|)$.

**Example 4.** *Figure 5.1 illustrates an example of the* Vari *method. The full sequence* $L$ *is* $\{15, 17, 18, 19, 20, 23, 33, 37, 39, 40, 4058, 4152, 4156, 4230, 4235\}$. *Assuming that the buffer size is* 12 *and that* 11 *elements are already present in the buffer, when element* 4152 *arrives and the buffer is full, a dynamic programming process is performed on the buffer, and the first ten elements are compressed into a block. Elements* $\{4058, 4152\}$ *are then appended to the buffer and wait for the next compression operation. Finally,* $B_1.n_1$ *is calculated to be* $\lceil \log(25 + 1) \rceil = 5$, *and* $B_2.n_2$ *is determined to be* $\lceil \log(177 + 1) \rceil = 8$. *The total number of bits in* $B_1$ *and* $B_2$ *are* $5 \times 9 = 45$ *and* $8 \times 4 = 32$, *respectively. Thus,* Vari *requires* $69 \times 2 + 45 + 32 = 215$ *bits to represent the entire list, while the original size is* $32 \times 15 = 480$ *bits. The compression ratio achieved by* Vari *in this example is* $\frac{480}{215} = 2.23$. *Note that, on the other hand, the* Fix *approach needs* 294 *bits with a compression ratio of* 1.63.

Note that the peak memory usage of the above methods is based on the size of the uncompressed region and the size of the compressed index at that time point. As shown above, the space overhead of the former one is rather limited. Therefore, the overhead encountered by these two approaches to support the online index compression is rather trivial.

## 5.3 Benefit Estimation Model based Algorithm

We observe that both the online version of fix-length and variable-length compression approaches have certain shortcomings. More precisely, the fix-length method is efficient but has a limited compression ratio, whereas the variable-length method demonstrates significant compression capabilities but incurs considerable time overhead during the compression process. In addition, both approaches require users to manually identify the maximum cardinality of the uncompressed region, which can prove challenging to optimize for distinct workloads and datasets and may even be unknown for previously unseen data.

To achieve a trade-off between them and avoid having to tune the hyper-parameter, we

propose and employ a new model that estimates the benefits of generating blocks.

The main idea is that the decision of whether to move elements from uncompressed regions to compressed blocks should be based on the number of bits that we can expect to save. In the online scenario, we do not know the whole sequence of incoming elements and, therefore, we need to use a probabilistic model to predict it.

We treat the gaps between every two consecutive elements from the input sequence of size $M$ as a probability distribution [1].

In our specific context, we have employed Kernel Density Estimation [9] (KDE) to approximate the probability density function (PDF) $f(x)$ of the inter-element gaps in the buffer. To this end, we use a kernel function $K(\cdot)$to determine the shape of the estimated density function, and a smoothing parameter, or bandwidth, $H(\cdot)$, to control the degree of smoothing in the estimated density function. We finally derive our estimated PDF as Equation (5.7). Note that following existing solutions such as [8], we could not only calculate our PDF based on the already observed elements but also update our PDF incrementally upon the arrival of new elements.

$$f(x) = \sum_{i=0}^{n-1} \frac{1}{H(x_i)} K(\frac{x - x_i}{H(x_i)}) \tag{5.7}$$

where $x_i$ represents the gap between elements $Z_i$ and $Z_{i+1}$; $K(\cdot)$ is a kernel function, and the Epanechnikov Kernel is used here.

Based on the above PDF, the incoming sequences can then be predicted by estimating the Cumulative Distribution Function (CDF) of the gap. This can be done by making an Inverse Sampling on $f(x)$ [19]: if currently there are $k$ elements, the $k+1$-th element can be computed iteratively with Equation (5.8).

$$Z_{k+1} = Z_k + \lceil [\int_{-\infty}^{u} f(u)du]^{-1} \rceil \tag{5.8}$$

[1]According to the findings in Theorem 1, the maximum number of $M$ is 138.

where $u$ is a random number from the standard uniform distribution in the interval $[0, 1]$ (e.g., from $U \sim \text{Uniform}[0, 1]$.)

Following this route, suppose the first element in current uncompressed region is $Z_0$, the benefit $G(Z_k)$ of compressing $Z_0$ through $Z_k$ can be calculated as follows:

$$G(Z_k) = \Theta_k - (\delta + b_k * k) \tag{5.9}$$

where $\Theta_k$ is the size in bits of corresponding uncompressed block; $\delta$ is the overhead needed save the necessary meta-data of a block (which as illustrated in Section 4 is equal to 69).

Finally, the goal of this benefit model is to find a position $k$ in the estimated incoming sequence where the benefit $B(Z_k)$ is maximized as shown by Equation (5.10):

$$B(Z_k) = \frac{1}{M - n} \sum_{k=n+1}^{M} G(Z_k) \tag{5.10}$$

Although the above model is theoretically sound, it still incurs considerable overhead in maintaining the statistics. Thus we propose an adaptive compression approach, named Adapt, also inspired by the high-level idea of estimating benefits. More specifically, we make the decision based on ONE newly incoming element: when a new element arrives, it compares the bits saved by compressing all elements in the uncompressed region $U$ when the new element is present and when it is not. Since the metadata of a new block requires 69 bits while including an additional uncompressed element requires 32 bits, the initial benefit is $-\rho = 37$. We will use $\rho$ as a predicate to make the decision in our model.

Suppose that there are $x$ elements in $U$ and the number of bits each delta occupies is $\bar{b}$; then, the saved number of bits $b'$ is calculated as $(x - 1) * (32 - \bar{b}) - \rho$. Note that the time complexity for Adapt for each element will be $\mathcal{O}(1)$ because the calculation time of expected gains is constant. The process to deal with a new element is shown in Algorithm 3.

**Example 5.** *Given the same list L above, suppose we already have elements* $\{15, 17, 18, 19, 20, 23, 33, 37, 39, 40\}$ *in* $U$ *and the benefit* $b'$ *is* $((10 - 1) \times (32 - 5) - 37) = 206$ *because the*

**Algorithm 3**: Adaptive Compression Approach

**Input**: $e$: The input element; $U$: The uncompressed region; $I$: The compression region; $\rho = 37$: The initial gain

1 **begin**
2 Calculate the expected benefits $b'(b'')$ using all existing elements from $U$ without (with) $e$;
3 **if** $b' - b'' > \rho$ **then**
4 Compress all elements in $U$ and create a new data block;
5 Add the new data block into $I$ and clear all elements in $U$; Append $e$ into $U$;
6 **else**
7 Append $e$ into $U$;
8 **end**

*max delta takes* $\lceil \log(40-15+1) \rceil = 5$ *bits. For the* Adapt *algorithm, when* 4058 *comes in the max delta takes* $\lceil \log(4058 - 15 + 1) \rceil = 12$ *bits, and thus* Adapt *decides to open a new block because* $((10-1) \times (32-5) - 37) - ((11-1) \times (32-12) - 37) > \rho$*, which means it no longer benefits from appending* 4058 *to the same block in* $U$*, so we create a data block for the current* $U$ *and then empty it. The element* 4058 *is then appended to the* $U$ *for the next compression. When the last element* 4235 *arrives and we finish our string similarity join, we perform a final compression over* $U$ *and we have* 2 *data blocks* $B_1$ *and* $B_2$*, and thus two corresponding metadata blocks* $M_1$ *and* $M_2$*. The start values* $B_1$ *and* $B_2$ *are* 15 *and* 4058*. The actual data blocks are as follows:* $B_1 = \{2, 3, 4, 5, 8, 18, 22, 24, 25\}$ *and* $B_2 = \{94, 98, 172, 177\}$*. Therefore since* $B_1.n_1 = \lceil \log(25+1) \rceil = 5$ *and* $B_2.n_2 = \lceil \log(177+1) \rceil = 8$*, the total number of bits of* $B_1$ *and* $B_2$ *are* $5 \times 9 = 45$ *and* $8 \times 4 = 32$*, respectively. In total,* Adapt *uses* $45+32+69+69 = 215$ *bits to represent the whole list while the original size is* 480 *bits. The compression ratio of* Adapt *online is* $\frac{480}{215} = 2.23$*, i.e it achieves the same performance as* Vari.

# CHAPTER 6

# Discussion

In this section, we investigate the potential benefits of using the algorithms proposed in CSS in novel application scenarios created by emerging new hardware.

## 6.1 Extension for Offline Approaches

Firstly, we would like to show that the offline compression algorithm proposed in Section 4 could also be potentially applied to Solid State Drive (SSD) based settings. Compared with hard disks, SSD has two unique characteristics: (i) its speed of random read is similar to that of sequential read, and (ii) it supports parallel I/O operations. Therefore, unlike the situation for hard disk, the two-layer index structure utilized by CSS dovetails with SSD. Assuming that the compressed inverted lists are stored on SSD, then when we need to look up a particular list, we can still use the binary search algorithm on both metadata and data blocks in the same way we did with the in-memory setting. This is due to the fast speed of random read operations on SSD. Moreover, as the index is constructed in the offline step and dumped to SSD at once, this will not incur expensive random write operations. As a result, the proposed algorithm can still be used on SSD without losing its main benefits.

## 6.2 Online Approaches

In this subsection, we discuss how to extend our online compression algorithms of CSS to make it cache-aware (Section 6.2.1) and SIMD-aware (6.2.2) in order to further improve

performance.

### 6.2.1 Cache-aware design

Here we discuss how to extend our online compression algorithms to be cache-aware. The goal is to maximize cache hits by ensuring that a cache line brought from memory to CPU caches (i.e., L1/L2/L3 cache) is fully utilized before it is retired. When a CPU instruction encounters a memory access, it first checks whether the accessed data is kept in caches. If so, then then we have a *cache-hit*. Otherwise, a cache line (64 bytes) of data is loaded from the main memory to caches. Loading data from main memory to caches decreases performance. A cache-aware design seeks to maximize the cache-hit rate. To convert the online compression algorithms to cache-aware design, the main idea is to organize the elements in metadata blocks and data blocks into a B-tree structure with the node size being a CPU cache line (64 bytes) [22, 41]. Instead of using tree pointers, the B-tree is materialized as an array supporting a level-order traversal manner, whereby search can be done efficiently by traversing the B-tree. One remaining challenge is that a number of elements is required to form a perfect tree; in fact, $17^i - 1$ elements are needed to form a $i$-level perfect tree. To reduce the space overhead, we can use optimizations that convert the array of the sorted elements to a complete tree instead of a perfect tree [41].

### 6.2.2 SIMD-aware design

Here we discuss how to further improve the query processing performance by using SIMD (i.e., Single instruction, multiple data) instructions. A SIMD instruction operates on an $r$-bit register where $r$ is 128 or 256, depending on different processors. The benefit of using SIMD instructions is the ability of processing multiple elements at a time. To organize the data elements in metadata blocks and data blocks into a SIMD-efficient structure, we need to store the data as follows:

(i) The elements in metadata blocks are first stored as a sequence of cache lines. Then elements in the same cache line are organized using k-ary method [38]. With this data organization, a SIMD operation can be applied to find the right child node to access in an efficient manner;

(ii) For the elements in data blocks, we just need to keep the same organization as that for the cache-aware design above. A SIMD operation interacts with a $r$-bit SIMD register as a vector of banks, where a bank is a continuous section of $b$ bits. [1] Given a search key, CSS first accesses the metadata layer in a SIMD-aware manner, as described in [38], to (a) find the data block containing the key, and then (b look up the key within it.

[1] $b$ can be 8 (in byte type), 16 (in short type), or 32 (int type).

# CHAPTER 7

# Evaluation

In this chapter, we conduct an extensive set of experiments to validate our proposed techniques.

## 7.1 Experiment Setup

Table 7.1: Statistics of Datasets

| Name | Average Length | Cardinality | Size (MB) |
|---|---|---|---|
| DBLP | 12.1 | 10 M | 155 |
| Tweet | 21.6 | 2 M | 203.3 |
| DNA | 103 | 1 M | 269.9 |
| AOL | 20.9 | 1.2 M | 27.6 |

We evaluate our proposed techniques on four real-world datasets which have been widely used in previous studies about string similarity search and join. DBLP [1] is a computer-science bibliographic dataset. We tokenize each record into a set of 3-grams. Tweet [2] consists of posts on the Twitter website. We split each record into tokens using the space as thee delimiter. DNA [3] is a dataset of DNA sequences. We tokenize each record into a set of

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

[2]https://twitter.com/

[3]https://www.ncbi.nlm.nih.gov/genome

6-grams. AOL [4] is a set of query logs. We use AOL for experiments with Edit Distance as the similarity metrics while using the other three datasets for Jaccard.

Table 7.1 provides comprehensive statistics on all datasets. Notably, we employed distinct length definitions for the AOL dataset and the other datasets. More specifically, for the AOL dataset, length refers to the number of characters in each string, while for the other datasets, length denotes the number of tokens.

To evaluate the generality of CSS, we test them on several different string similarity search and string similarity join frameworks. For similarity search, we focus on the $T$-Occurrence problem and evaluate the ScanCount and MergeSkip proposed in [26]. For similarity join, we evaluate the Count Filter [21], the Prefix Filter [11] and the Position Filter [46] for Jaccard similarity and the Segment Filter [28] for Edit Distance. Although there are also many other filtering techniques [37, 40, 18, 45], we omit them in the experiments due to space limitations. Since they are all variants of the above approaches, CSS can also be seamlessly integrated into them. Our experimental focus primarily centers on evaluating the index size of the proposed compressed index structures. Furthermore, we compare the query execution time of all methods with those obtained using uncompressed inverted lists.

All experiments are conducted on a server with an Intel Xeon(R) CPU processor, 16 GB RAM, running Ubuntu 14.04.1. For the fairness of comparing memory overhead, we implement all methods by ourselves. All the algorithms are implemented in C++ and compiled with GCC 4.8.4.

## 7.2 Effect of Compression Techniques

We first look at the index size of each method. The results of the similarity search are detailed in Table 7.2, wherein we implement four distinct approaches. Firstly, Uncomp represents the method that utilizes the original inverted lists without any form of compression. Secondly,

[4] https://jeffhuang.com/search_query_logs.html

Table 7.2: Index Size for Compression Schemes: Similarity Search (MB)

| Dataset | Uncomp | PForDelta | MILC | CSS |
|---|---|---|---|---|
| DBLP | 992.68 | 496.45 | 229.26 | 200.10 |
| Tweet | 351.92 | 186.24 | 107.55 | 85.84 |
| DNA | 1812.76 | 1020.30 | 408.06 | 376.66 |
| AOL | 191.80 | 96.06 | 44.31 | 40.2 |

PForDelta [51] and MILC [41] represent state-of-the-art inverted list compression methods. Thirdly, we present CSS as the variable-length approach in Section 4. Our results indicate that, through the application of the compression scheme, CSS is able to significantly reduce the memory space requirements for inverted lists. For instance, on the DNA dataset, the compression ratios of MILC and CSS are 4.44 and 4.82, respectively. Moreover, we observe that our proposed methods, compared to the existing compression scheme PForDelta, have considerably lower memory consumption.

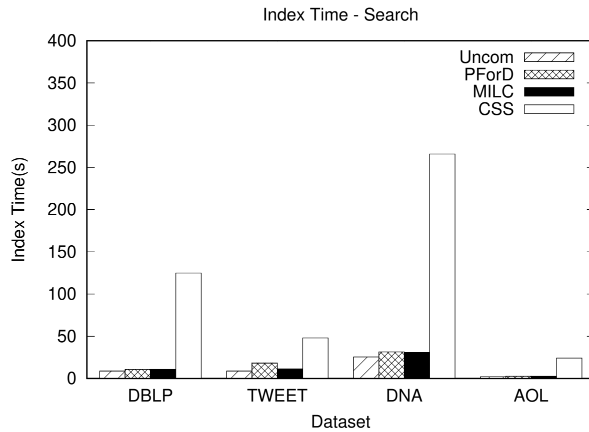



Figure 7.1: Index Time for Similarity Search

Figure 7.1 displays the index time of all the compression schemes under consideration. Notably, we observe that the indexing time of MILC is similar to that of Uncomp. Although the dynamic programming algorithm employed by CSS incurs an index construction over-

head, this overhead is acceptable since the compressed index is constructed in the offline step.

Table 7.3: Index Size for Compression Schemes: Similarity Join (MB)

| Dataset | Uncomp | Fix | Vari | Adapt |
|---|---|---|---|---|
| DBLP | 992.68 | 361.48 | 201.45 | 225.36 |
| Tweet | 147.61 | 59.69 | 44.56 | 45.73 |
| DNA | 554.70 | 260.75 | 188.94 | 192.61 |
| AOL | 72.22 | 34.91 | 29.94 | 40.76 |

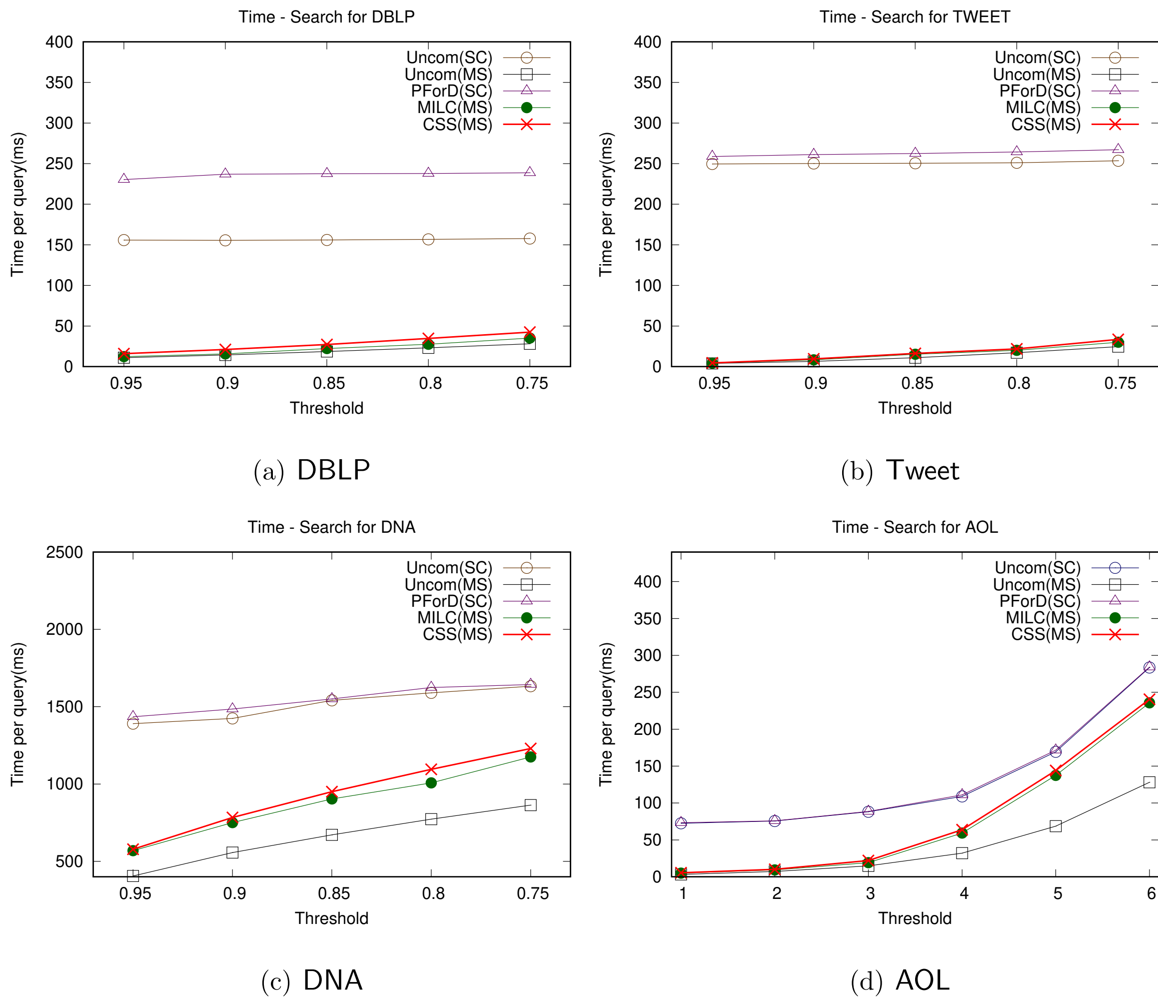
Due to space constraints, we report the results of only one existing filtering technique for similarity join on a single dataset. However, we note that this technique exhibits similar trends when evaluated on different combinations of filter and dataset. In particular, we evaluate the Count Filter on the DBLP dataset, the Prefix Filter on the Tweet dataset, the Position Filter on the DNA dataset, and the Segment Filter on the AOL dataset. Notably, different thresholds in similarity join problems can result in varying index sizes. Thus, we report our results for a Jaccard threshold of 0.6 on the first three datasets and an edit distance threshold of 4 on the AOL dataset.

We implement the following four methods: Uncomp, which represents the original inverted list; Fix and Vari, which are online algorithms extended from MILC and CSS, respectively; and Adapt, which is the adaptive compression method introduced in Section 5.3. Based on the results shown in Table 7.3, we make the following observations. Firstly, in the case of the Count Filter on the same dataset DBLP, the compression ratio achieved is not as significant as that of the corresponding offline compression schemes. This can be attributed to the inability of online algorithms to use information from the entire list when making decisions on compressed blocks. Nonetheless, the online algorithm still achieves a reasonable compression ratio. For instance, on the DBLP dataset, the compression ratios of Fix, Vari, and Adapt are 2.75, 4.93, and 4.40, respectively. Secondly, Vari exhibits the highest compression ratio since

it performs dynamic programming on the subsequence obtained so far during compression. However, Vari also suffers from additional overhead in execution time, as demonstrated in the subsequent subsection. Thirdly, the compression ratio achieved by Adapt is very close to that of Vari. For instance, on the Tweet dataset, the compression ratios of Vari and Adapt are very similar, with values of 3.31 and 3.23, respectively. This demonstrates that the approximation made in Adapt is reasonable and can provide a good trade-off to save memory using the high-level idea inspiring our benefit model.

## 7.3 End-to-end Query Time

We can now examine the end-to-end query execution time of the above-mentioned approaches. For the string similarity search, we randomly select 10,000 strings from each dataset as queries and report the average time per query. In particular, we utilize the Edit Distance as the similarity metric for AOL, while employing Jaccard for the remaining three datasets. We implement five methods, namely, the ScanCount (SC) algorithm performed on Uncomp and PForDelta inverted lists, the MergeSkip (MS) algorithm performed on Uncomp, MILC, and CSS inverted lists. The results are presented in Figure 7.2. However, since PForDelta does not support random access to lists, the more efficient MergeSkip algorithm cannot be performed on it. Therefore, the performance of similarity search on PForDelta is impacted by the low efficiency of ScanCount. We observe that the performance of MergeSkip on MILC and CSS is very similar to that of the uncompressed inverted lists. For instance, for $\tau = 0.75$ on the Tweet dataset, the average search time for MergeSkip on Uncomp, MILC, and CSS is 24.6, 30, and 33.6 milliseconds, respectively. The reason that search on MILC has better performance than CSS can be attributed to the fact CSS would use more blocks in most cases. As a result, there will be more overhead for the binary search. This is more obvious in Figure 7.2(c) and 7.2(c) as the skewness of token distribution is more obvious due to the characteristics of datasets. As a result, there are longer inverted lists, and the extra

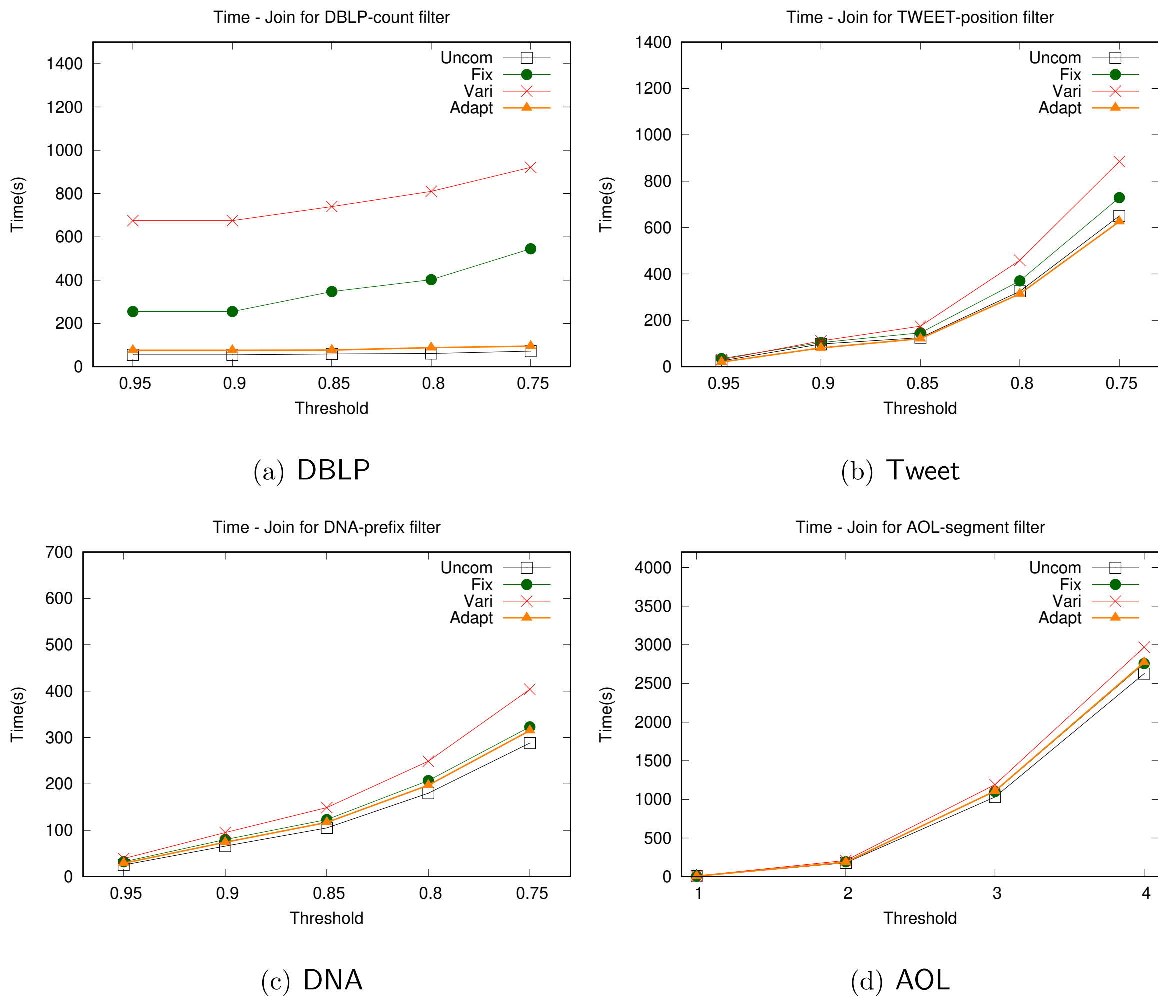



(a) DBLP (b) Tweet (c) DNA (d) AOL

Figure 7.2: Comparison of Execution Time: Similarity Search

binary search operations on them can lead to more overhead in search time. Also note that due to the attributes of AOL, when we relax the edit-distance threshold, the execution time for all method, including Uncomp, grow faster due to the increasing search space.

The results of similarity join are presented in Figure 7.3. In most cases, the performance on compressed inverted lists is very similar to that on uncompressed ones. For instance, for a Jaccard threshold of $\tau = 0.8$ on the DNA dataset, the join time of the Prefix Filter algorithm on the uncompressed dataset is 180 seconds, while the time on Fix, Vari, and Adapt is 207, 249, and 197 seconds, respectively. We observe that Vari exhibits the worst performance

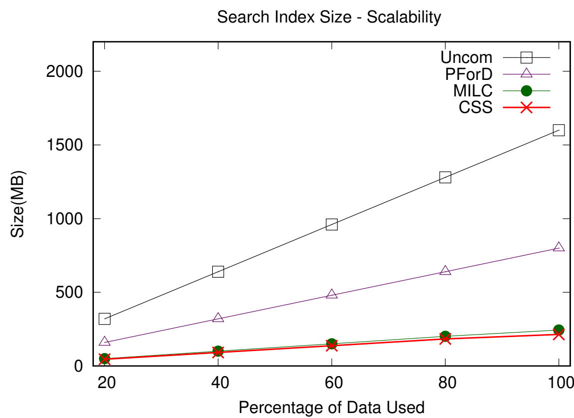



(a) DBLP (b) Tweet

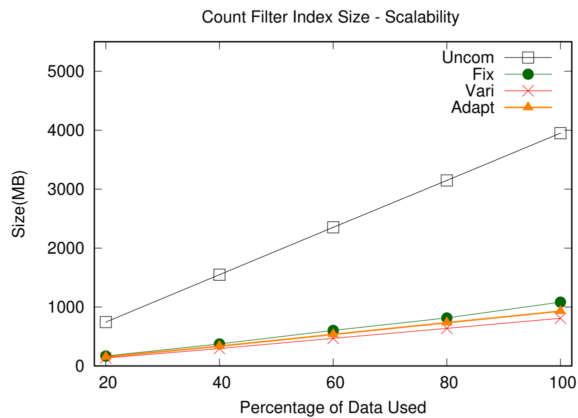
(c) DNA (d) AOL

Figure 7.3: Comparison of Execution Time: Similarity Join

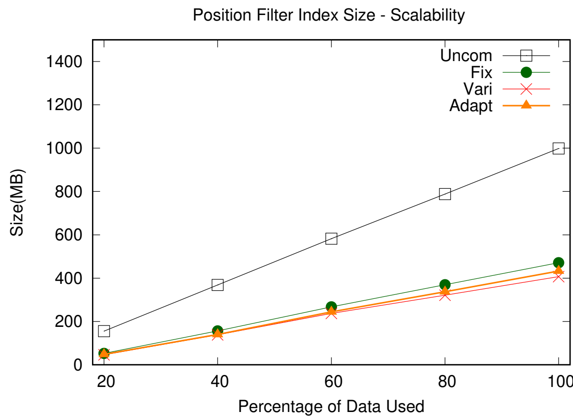
under all settings due to the cost of the required dynamic programming calculations, which is reasonable considering the high space efficiency delivered by this additional time overhead. Meanwhile, the performance of Adapt is very similar to that of Uncomp, but it utilizes much less memory space. Due to its fast bit operations, Adapt can even outperform Uncomp in some cases. For instance, when $\tau = 0.8$ on the Tweet dataset, the join time for the Position Filter on Uncomp is 325 seconds, while the time on Adapt is 314 seconds. Hence, we can conclude that Adapt represents an ideal choice for supporting similarity join applications.




(a) Similarity Search




(b) Similarity Join: Count Filter

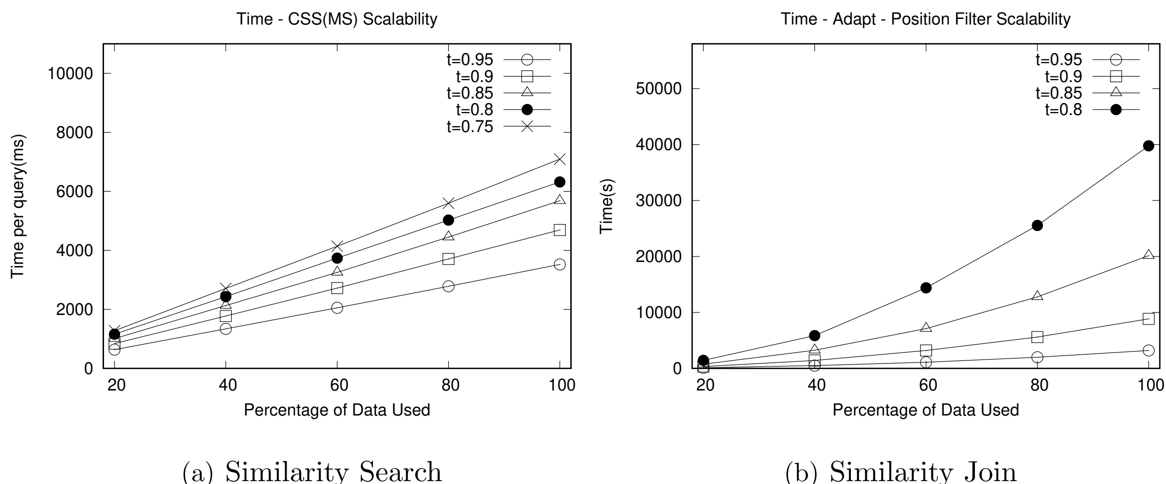



(c) Similarity Join: Position Filter

Figure 7.4: Scalability: Index Size

## 7.4 Scalability

In addition, we evaluate the scalability of our proposed method. We use the generator proposed in [34] to generate datasets for the experiment. Here we create two synthetic datasets: the first dataset following the Zipf distribution has an average set size of 50 and a universe size of 116346; the other dataset following the Uniform distribution has an average set size of 25, and a universe size of 150. The cardinality of both datasets is 10 million. For similarity search, we evaluate the four compression schemes on the Uniform synthetic data;




(a) Similarity Search

(b) Similarity Join

Figure 7.5: Scalability: Execution Time

For similarity join, we evaluate the Position Filter and Count Filter performed on Adapt compression scheme on the Zipf data.

The results of memory usage for datasets with different sizes are shown in Figure 7.4. We can see that it achieves linear scalability for both similarity search and join algorithms. For example, on Uniform synthetic data, when the size of dataset scales from 20% to 100%, the corresponding index size for CSS is 45.78, 91.66, 137.57, 183.49, and 214.36 MB, respectively.

Additionally, we report the query execution time for our proposed methods. Specifically, we utilize the MergeSkip algorithm on the CSS compression scheme for similarity search, while we use the Position Filter on the Adapt compression scheme for similarity join. The scalability results for both similarity search and similarity join are presented in Figure 7.5. We observe that our methods exhibit good scalability for similarity search, while for similarity join, the scalability is quadratic, consistent with the increasing search space.

Table 7.4: Index Size: Amazon Review

(a) Similarity Search

| Schemes | Size (GB) |
|---|---|
| Uncomp | 39.4 |
| PForDelta | 18.7 |
| MILC | 8.7 |
| CSS | 7.9 |

(b) Similarity Join

| Schemes | Size (GB) |
|---|---|
| Uncomp | 39.4 |
| Fix | 11.9 |
| Vari | 8.1 |
| Adapt | 8.9 |

## 7.5 Case Study

In order to demonstrate the benefits of the memory savings brought by our proposed methods, we conduct a case study utilizing a dataset Amazon Reviews dataset [35]. This is a large-scale dataset that has been frequently used as a benchmark for big data systems [23]. As our work is focused on the environment of a single machine, we use the 5-core review data set. The size of the raw review text data is approximately 7GB.

The results of index size for different compression mechanisms are presented in Table 7.4. We observe that the index size of the uncompressed index and PForDelta for similarity search is 39.4 GB and 18.7 GB, respectively, which exceeds the available memory size of 16 GB. Consequently, it is necessary to process it with expensive disk-based algorithms. In contrast, the index size of our best compression mechanism, CSS, is just 7.9 GB. Therefore, we can utilize an in-memory algorithm over compressed indexes. Similar trends also occur in the application of similarity join. Consequently, our proposed algorithms can save significant memory space in large-scale applications, avoiding the need for expensive disk-based algorithms.

# CHAPTER 8

# Related Work

### 8.0.1 String Similarity Search and Join

String similarity search and string similarity join has been a hot topic in the database community over the past decades. A mainstream of existing solutions is to develop different filtering techniques to reduce the number of candidates to be verified. Gravano et al. [21] proposed the Count Filter, where strings are regarded as candidates when they share enough common signatures. Li et al. [26] devised several list merging algorithms to reduce the filtering cost of Count Filter. Chaudhuri et al. [11], and Bayardo et al. [7] developed the idea of the Prefix Filter: two strings are similar only when they share at least one common signature in their prefixes. Xiao et al. [46] proposed the Position Filter based on the Prefix Filter by taking the position of signatures into consideration when generating candidates. Xiao et al. [45] and Qin et al. [37] followed this route to reduce the prefix length further to improve the performance. The Segment Filter [18, 28] adopts the pigeonhole theory and utilizes disjoint segments as signatures, which significantly improves the filter power.

### 8.0.2 Data Compression

Data compression techniques have been widely adopted in many data-management-related studies, such as web search [49], machine learning [27], column database [4], time series processing [29] and graph analysis [30].

The compression techniques in column databases can be classified into two categories:

heavy-weighted compression schemes and light-weighted compression schemes. Heavy-weighted compression schemes [50, 13] have prohibitively expensive decompression costs. To access an element, they need to decompress the whole compressed data. Light-weighted compression schemes, e.g., Dictionary based Encoding methods [12, 43] and Run-length Encoding [30], support directly querying over compressed data. The common design goal of the above methods is to save space for duplicated elements. Since the inverted lists in string similarity search and join have no duplicate elements, the above schemes cannot work well.

Compression techniques in Information Retrieval are designed to minimize the space overhead of inverted lists. Most existing compression methods save space by leveraging the deltas between elements. Examples include PForDelta [51] and its variants [49, 24], VB [15], GroupVB [16], Simple8b [5] and PEF [36]. One problem is that they have to decompress the whole list for query processing, and MILC [41] addresses this problem by using a two-level storage structure. Since the above methods must construct the index in the offline step, they cannot support string similarity join.

Time series compression schemes are usually lossy techniques, which cannot be applied in our cases where lossless is a mandatory requirement. The most common approach is to approximate the data as a sequence of low-order polynomials [20]. An alternative is to discretize the time series using Symbolic Aggregate Approximation [31] or its variations [39].

# CHAPTER 9

# Conclusion

In this thesis, we propose a unified compression framework that is capable of supporting a wide range of string similarity search and string similarity join frameworks. Thus, we present a comprehensive set of fundamental list operations for string similarity search and string similarity join that can be directly applied to compressed inverted lists and develop effective compression techniques that are tailored to these operations.

To support similarity join, we introduce the first framework that can construct compressed inverted lists under online settings and devise a benefit model to provide guidance for compression strategies. Building on this, we further develop an adaptive approach that aims to balance the time and space overhead.

Experimental results on real-world datasets demonstrate that our approach can significantly reduce memory consumption while achieving query performance similar to that of existing techniques that use uncompressed indexes. It should be noted that our online compression algorithms can be applied to other problems that require on-the-fly list construction and list operations, such as time series matching and DNA sequence comparisons.

# REFERENCES

[1] BZIP Compression. `http://www.bzip2.org/`.

[2] LZO Compression. `http://www.oberhumer.com/opensource/lzo/`.

[3] ZLIB Compression. `http://www.zlib.net/`.

[4] D. J. Abadi, S. Madden, and M. Ferreira. Integrating compression and execution in column-oriented database systems. In *SIGMOD*, pages 671–682, 2006.

[5] V. N. Anh and A. Moffat. Index compression using 64-bit words. *Softw. Pract. Exp.*, 40(2):131–147, 2010.

[6] R. A. Baeza-Yates, C. Castillo, F. Junqueira, V. Plachouras, and F. Silvestri. Challenges on distributed web retrieval. In *ICDE*, pages 6–20, 2007.

[7] R. J. Bayardo, Y. Ma, and R. Srikant. Scaling up all pairs similarity search. In *WWW*, pages 131–140, 2007.

[8] A. P. Boedihardjo, C. Lu, and F. Chen. Fast adaptive kernel density estimator for data streams. *Knowl. Inf. Syst.*, 42(2):285–317, 2015.

[9] Z. Botev, J. F. Grotowski, and D. P. Kroese. Kernel density estimation via diffusion. *The annals of Statistics*, 38(5):2916–2957, 2010.

[10] S. Chambi, D. Lemire, O. Kaser, and R. Godin. Better bitmap performance with roaring bitmaps. *Softw. Pract. Exp.*, 46(5):709–719, 2016.

[11] S. Chaudhuri, V. Ganti, and R. Kaushik. A primitive operator for similarity joins in data cleaning. In *ICDE*, page 5, 2006.

[12] Z. Chen, J. Gehrke, and F. Korn. Query optimization in compressed database systems. In *SIGMOD*, pages 271–282. ACM, 2001.

[13] K. Chung and J. Wu. Level-compressed huffman decoding. *IEEE Trans. Communications*, 47(10):1455–1457, 1999.

[14] A. Colantonio and R. D. Pietro. Concise: Compressed 'n' composable integer set. *Inf. Process. Lett.*, 110(16):644–650, 2010.

[15] D. R. Cutting and J. O. Pedersen. Optimizations for dynamic inverted index maintenance. In *SIGIR*, pages 405–411. ACM, 1990.

[16] J. Dean. Challenges in building large-scale information retrieval systems: invited talk. In *WSDM*, page 1. ACM, 2009.

[17] F. Deliège and T. B. Pedersen. Position list word aligned hybrid: optimizing space and performance for compressed bitmaps. In *EDBT*, volume 426, pages 228–239, 2010.

[18] D. Deng, G. Li, H. Wen, and J. Feng. An efficient partition based method for exact set similarity joins. *PVLDB*, 9(4):360–371, 2015.

[19] L. Devroye. Nonuniform random variate generation. *Handbooks in operations research and management science*, 13:83–121, 2006.

[20] F. Eichinger, P. Efros, S. Karnouskos, and K. Böhm. A time-series compression technique and its application to the smart grid. *VLDB J.*, 24(2):193–218, 2015.

[21] 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.

[22] C. Kim, J. Chhugani, N. Satish, E. Sedlar, A. D. Nguyen, T. Kaldewey, V. W. Lee, S. A. Brandt, and P. Dubey. FAST: fast architecture sensitive tree search on modern cpus and gpus. In *SIGMOD*, 2010.

[23] T. Kim, W. Li, A. Behm, I. Cetindil, R. Vernica, V. R. Borkar, M. J. Carey, and C. Li. Supporting similarity queries in apache asterixdb. In *EDBT*, pages 528–539, 2018.

[24] D. Lemire and L. Boytsov. Decoding billions of integers per second through vectorization. *Softw. Pract. Exp.*, 45(1):1–29, 2015.

[25] D. Lemire, O. Kaser, and K. Aouiche. Sorting improves word-aligned bitmap indexes. *Data Knowl. Eng.*, 69(1):3–28, 2010.

[26] C. Li, J. Lu, and Y. Lu. Efficient merging and filtering algorithms for approximate string searches. In *ICDE*, pages 257–266, 2008.

[27] F. Li, L. Chen, Y. Zeng, A. Kumar, X. Wu, J. F. Naughton, and J. M. Patel. Tuple-oriented compression for large-scale mini-batch stochastic gradient descent. In *SIGMOD*, pages 1517–1534, 2019.

[28] G. Li, D. Deng, J. Wang, and J. Feng. Pass-join: A partition-based method for similarity joins. *PVLDB*, 5(3):253–264, 2011.

[29] C. Lin, E. Boursier, and Y. Papakonstantinou. Approximate analytics system over compressed time series with tight deterministic error guarantees. *PVLDB*, 13(7):1105–1118, 2020.

[30] C. Lin, B. Mandel, Y. Papakonstantinou, and M. Springer. Fast in-memory SQL analytics on typed graphs. *PVLDB*, 10(3):265–276, 2016.

[31] J. Lin, E. J. Keogh, S. Lonardi, and B. Y. Chiu. A symbolic representation of time series, with implications for streaming algorithms. In *DMKD@SIGMOD*, pages 2–11. ACM, 2003.

[32] J. Lu, C. Lin, J. Wang, and C. Li. Synergy of database techniques and machine learning models for string similarity search and join. In *CIKM*, pages 2975–2976, 2019.

[33] J. Lu, C. Lin, W. Wang, C. Li, and H. Wang. String similarity measures and joins with synonyms. In *SIGMOD*, pages 373–384, 2013.

[34] W. Mann, N. Augsten, and P. Bouros. An empirical evaluation of set similarity join techniques. *PVLDB*, 9(9):636–647, 2016.

[35] J. J. McAuley, R. Pandey, and J. Leskovec. Inferring networks of substitutable and complementary products. In *ACM SIGKDD*, pages 785–794, 2015.

[36] G. Ottaviano and R. Venturini. Partitioned elias-fano indexes. In *SIGIR*, pages 273–282. ACM, 2014.

[37] J. Qin, W. Wang, Y. Lu, C. Xiao, and X. Lin. Efficient exact edit similarity query processing with the asymmetric signature scheme. In *SIGMOD*, pages 1033–1044, 2011.

[38] B. Schlegel, R. Gemulla, and W. Lehner. k-ary search on modern processors. In *DaMoN*, pages 52–60, 2009.

[39] J. Shieh and E. J. Keogh. *i*sax: disk-aware mining and indexing of massive time series datasets. *DMKD*, 19(1):24–57, 2009.

[40] J. Wang, G. Li, and J. Feng. Can we beat the prefix filtering?: an adaptive framework for similarity join and search. In *SIGMOD*, pages 85–96, 2012.

[41] J. Wang, C. Lin, R. He, M. Chae, Y. Papakonstantinou, and S. Swanson. MILC: inverted list compression in memory. *PVLDB*, 10(8):853–864, 2017.

[42] J. Wang, C. Lin, M. Li, and C. Zaniolo. An efficient sliding window approach for approximate entity extraction with synonyms. In *EDBT*, pages 109–120, 2019.

[43] J. Wang, C. Lin, Y. Papakonstantinou, and S. Swanson. An experimental study of bitmap compression vs. inverted list compression. In *SIGMOD*, pages 993–1008, 2017.

[44] J. Wang, C. Lin, and C. Zaniolo. Mf-join: Efficient fuzzy string similarity join with multi-level filtering. In *ICDE*, pages 386–397, 2019.

[45] C. Xiao, W. Wang, and X. Lin. Ed-join: an efficient algorithm for similarity joins with edit distance constraints. *PVLDB*, 1(1):933–944, 2008.

[46] C. Xiao, W. Wang, X. Lin, and J. X. Yu. Efficient similarity joins for near duplicate detection. In *WWW*, pages 131–140, 2008.

[47] G. Xiao, J. Wang, C. Lin, and C. Zaniolo. Highly efficient string similarity search and join over compressed indexes. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*, pages 232–244, 2022.

[48] H. Yan, S. Ding, and T. Suel. Inverted index compression and query processing with optimized document ordering. In *WWW*, pages 401–410, 2009.

[49] J. Zhang, X. Long, and T. Suel. Performance of compressed inverted list caching in search engines. In *WWW*, pages 387–396. ACM, 2008.

[50] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Trans. Inf. Theory*, 23(3):337–343, 1977.

[51] M. Zukowski, S. Héman, N. Nes, and P. A. Boncz. Super-scalar RAM-CPU cache compression. In *ICDE*, page 59, 2006.