DEVELOPING 3-IN-1 INDEX STRUCTURES ON COMPLEX STRUCTURE SIMILARITY SEARCH

WANG XIAOLI

(B. Eng., Northeastern University, China)

A THESIS SUBMITTED

FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

SCHOOL OF COMPUTING

NATIONAL UNIVERSITY OF SINGAPORE

2013

DECLARATION

I hereby declare that the thesis is my original work and it has been written by me in its entirety. I have duly acknowledged all the sources of information which have been used in the thesis.

This thesis has also not been submitted for any degree in any university previously.

WANG XIAOLI
August 2013

ACKNOWLEDGEMENTS

First and foremost, I would like to express my sincerest gratitude to my supervisor Assoc. Prof. Anthony K. H. Tung, who has supported me throughout my Ph.D study and research at National University of Singapore, for his patience, enthusiasm, and immense knowledge. Without his guidance, this dissertation would not have been completed or written.

My supervisor Anthony K. H. Tung has served as a life mentor. He has shared with me his invariable experience in both research and life, and guided me to work with an appropriate and positive attitude. My another project supervisor Prof. Beng Chin Ooi also deserves my deepest gratitude. His selfless sharing of work experience and life attitude can benefit my whole life. I would like to thank the rest members of the supervisory committee, Prof. Kian-Lee Tan and Prof. Wing-Kin Sung. Without their insightful comments, immense knowledge and kind assistance, this study would not have been successful. My sincere thanks also goes to Prof. Chee-Yong Chan, for offering me the job as a teaching assistant in his course, which helps to empower my teaching skill and speaking ability. I appreciate the efforts from all the collaborators in the past papers, including Assoc. Prof. Xiaofeng Ding, Ms. Shanshan Ying, Dr. Zhenjie Zhang, Assist. Prof. Sai Wu, Dr. Chuitian Rong, Mr. Sheng Wang, Dr. Wei Lu, Assoc. Prof. Yueguo Chen, Prof. Xiaoyong Du, and Prof. Hai Jin. I would like to thank Prof. H. V. Jagadish and prof. Ambuj K. Singh for their broad knowledge, care and patience throughout the discussions we had.

My sincerest gratitude goes to my roommates: Jia Hao, Meiyu Lu, and Meihui Zhang. In the past three years, they have always been there. My special thanks also goes to all my graduate friends, especially the database group fellows. In particular, I am grateful to Shaohuan Li, Tuo Liu, Weiran Liu, and Zhixing Yang. Their passions and ambitions on the social reading project encourage me and make me active. I am also very grateful to the workmates in the SESAME center, especially Ms. Adele Chen Zimmermann and Dr. Lekha Chaisorn. They are so kind and helpful to assist my work in the SESAME center. I would also like to convey thanks to the School of Computing for providing the financial means and laboratory facilities. Meanwhile, I am also grateful to the stuff of SOC, for providing helpful assistances.

I would like to thank my family. My parents Qingxian Wang and Xiufeng Wang, who gave birth to me at the first place, are supporting me faithfully and spiritually throughout my life. They give me selfless love whilst allowing me the room to study and work in my own way. My sisters Rongzhen Wang and Xiaoqin Chen, are always standing by me. Last but not the least, my deepest love goes to my dear husband, Yuewu Lin, for his long wait for me and continued support for my study and work.

CONTENTS

ABSTRACT

In traditional relational databases, data are modeled as tables. However, most real life data cannot be simply modeled as tables, but as complex structures like sequences, trees and graphs. Existing systems typically cater to the storage of complex structures separately. Therefore, each application domain may need to redesign the storage system for a specific complex structure. Obviously, this can result in a waste of resources. Moreover, many applications may require the storage of various complex structures, and it is not easy to adapt existing systems to support such applications. In this dissertation, we aim to develop a unified framework, denoted by 3-in-1, that can support the efficient storage and retrieval of various complex structures (i.e., sequences, trees, and graphs).

As graph is the most complex model, we first address the graph similarity search problem. A novel efficient indexing method is developed for handling graph range queries. In this method, a two-level inverted index is constructed based on the star decomposition method. Meanwhile, a set of effective and efficient pruning techniques are developed to support graph search. The proposed search algorithms follow a filter-and-refine framework. Comprehensive experiments on two real datasets show that the proposed method returns the smallest candidate set and outperforms all the state-of-the-art works. This is because the total query time can be reduced as much as possible as our method can significantly reduce the number of candidates for verification. Experimental results also show that our method takes reasonable filtering time compared with existing works. To extend the above inverted index structure to support efficient sequence similarity search, we then propose a novel pipeline

framework. We address the problem of finding k-nearest neighbors (KNN) in sequence databases, as this type of search is more general in real applications. Unlike most existing works which used short, exact n-gram matching in a filter-and-refine framework for approximate sequence search, our new approach allows us to use longer but approximate n-gram matching as a basis for pruning off KNN candidates. Based on this breakthrough, we adopt a pipeline framework over a two-level inverted index for searching KNN in the sequence database. By coupling this framework together with several efficient filtering strategies including the frequency queue and the well-known Combined Algorithm (CA), our proposal brings various enticing advantages over existing work, including progressive result update, early termination, and easily parallelization. With comprehensive experiments on three real datasets, the results show that our approach outperforms all the state-of-the-art works by achieving huge reduction on false positive candidates which will incur the expensive cost of verification.

We further investigate the problem of unified 3-in-1 indexing and processing for complex structures. From previous work, the inverted index has been shown to be effective to support efficient complex structure similarity search. Consequently, we use it as the basic index structure to develop a unified retrieval framework for supporting various complex structures. In this work, we implement the 3-in-1 system with three layers: the storage layer, the index layer, and the application layer. In the storage layer, various types of original data is stored in the file system. In the index layer, we implement a unified inverted index for various complex structures. The application layer is the processing layer where each type of complex structure can build specific processor to communicate with the other two layers. This system can be very useful as it can support many complex applications that involve a variety of complex structures. For instance, we apply it to a real ebook reading system for solving several real problems, and present the initial demo from `http://readpeer.com`.

LIST OF TABLES

LIST OF FIGURES

LIST OF SYMBOLS

D	a database of complex structures
q	a complex structure query like sequence, tree, or graph
g	a graph
E	the set of edges in a graph
$V(g)$	the set of vertices in a graph g
$deg(v)$	$\|\{u\|(u,v) \in E\}\|$, the degree of vertex v in a graph
$\delta(g)$	$\max_{v \in V(g)} deg(v)$
$\delta(D)$	$\max_{g \in D} \delta(g)$
$\lambda(g_1, g_2)$	the edit distance between graphs g_1 and g_2
$\lambda(st_1, st_2)$	the edit distance between stars st_1 and st_2
$\mu(g_1, g_2)$	the star mapping distance between g_1 and g_2
$\zeta(g_1, g_2)$	the overall score of g_2 obtained from g_1
s	a sequence
$\lambda(s_1, s_2)$	the edit distance between two sequences s_1 and s_2
$\lambda(ng_1, ng_2)$	the edit distance between two n-grams ng_1 and ng_2
$\mu(s_1, s_2)$	the gram mapping distance between two sequences s_1 and s_2
ϕ	the frequency threshold value of n-grams
τ	the edit distance threshold
$\tau(t)$	the threshold value computed by the CA aggregation function
$\eta(\tau, t, n)$	the number of n-grams affected by τ edit operations with gram edit distance $> t$

CHAPTER 1

Introduction

In the past decades, tremendous amount of data in various complex structures are collected and need to be managed. It is very important to model such data using appropriate data structures for storage. For example, in traditional data management system such as relational databases, data are modeled as tables. However, most complex data in the real world cannot be simply modeled as tables, but as complex structures like sequences, trees and graphs. For instance, real systems such as chemical compounds and web documents are often stored as graph structures in graph databases. The complex structure poses new challenging research problems that do not exist in traditional databases. In the literature, how to search the required and interesting complex objects has become an important research topic, and exiting work has focused on many related issues. Such issues are often presented as the complex structure search problems, such as the graph isomorphism problem, the string matching problem, the tree similarity search problem, and so on. The classical search problem is often formulated as the exact matching problem. However, in practice, extract matching is too restrictive, as real objects are often affected by noises. Therefore, complex structure similarity search has been attracting significant attention in many scientific fields for its general usage and wide applications.

1.1 Complex Models and Applications

To understand the importance of problems on complex structures, it is worthwhile to see the applications of complex structure models in practical research.

1.1.1 Graph Model and Search

Graph is a very powerful model. It has been applied to handling many interesting research problems in various domains including bio-informatics [30], chem-informatics [71], software engineering [18], pattern recognition [42], etc. Many researchers in these areas have used graph model to represent data and developed graph search algorithms to manage data. Figure 1.1 shows a series of interesting applications on graph models.

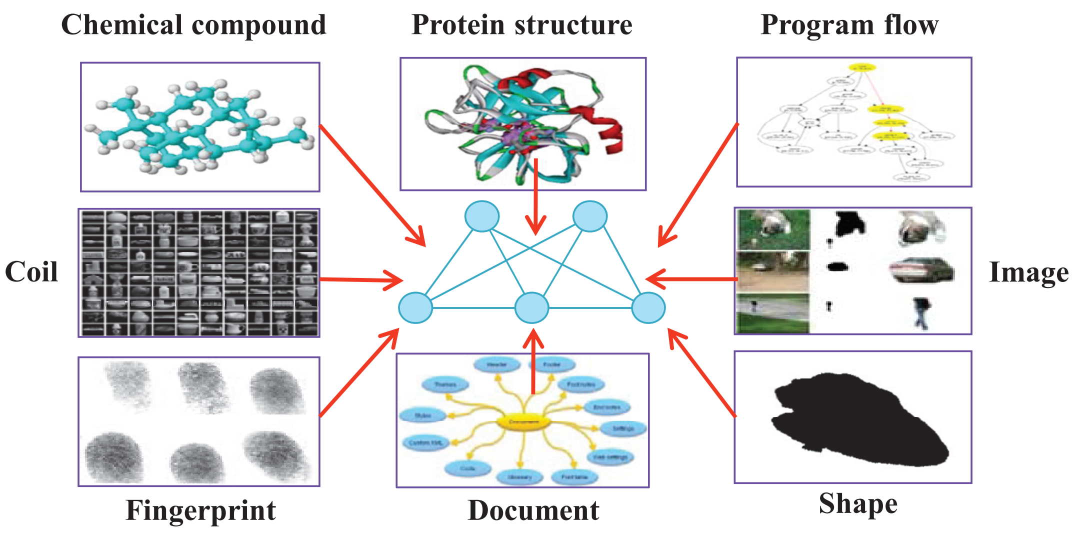


Figure 1.1: Examples on graph models

In bio-informatics and chem-informatics, graphs are usually used to model proteins and molecular compounds (e.g., [30, 71]). With the graph model, searching in protein databases helps to identify pathways and motifs among species, and assists in the functional annotation of proteins. Meanwhile, searching a molecular structure in a database of molecular compounds is useful to detect molecules that preserve chemical properties associated with a well-known molecular structure. This can be used in screening and drug design.

In software engineering, J. Ferrante et al. [18] used program dependence graph (PDG) to model the data flow and control dependency within a procedure. In a program dependence graph, vertices are statements and edges represent dependency between the statements. Searching in such program graph databases is widely applied to clone detection, optimization, debugging, etc (e.g., [19, 67]).

In pattern recognition, graphs have been shown to be efficient as a processing and representational scheme. There is a technical committee of the International Association for Pattern Recognition (IAPR)[1], dedicated to promote the graph research in this field. Specifically, Riesen K. et al. [50] collected graph databases with coils, fingerprints, web documents, etc. These databases have been used to do classification or search tasks for the graph research[2].

As listed above, it is essential to process graph searching efficiently for managing a large graph database. In particular, graph similarity search has been attracting more attention from researchers, as traditional exact matching problems (e.g., [22, 34]) is too restrictive to support the noise data in practice. This dissertation focuses on supporting similarity search in graph databases.

1.1.2 Sequence Similarity Search

Sequence has wide applications in a variety of areas including approximate keyword search [2], DNA/protein sequence search [44], plagiarism detection [51, 55, 81], ebook annotation search [68], etc. In the literature, numerous approximate string matching algorithms have been proposed to support the efficient sequence similarity search in the above applications.

Simply consider a keyword search example. A search engine may have to identify that names like "*E. L. Wood*" and "*Emma Louise Wood*" are potentially referring to the same person in the searching results.

In bio-informatics, it is important for solving problems like looking for given features in DNA chains or determining how different two genetic sequences are [32]. In such applications, exact matching is of little use. This is because queried gene sequence rarely matches existing gene sequences exactly: the experimental measures have errors of different kinds and even the correct chains may have small differences. Figure 1.2 shows a simple alignment example between two DNA sequences.

[1] `http://www.greyc.ensicaen.fr/iapr-tc15/index.php`
[2] `http://www.iam.unibe.ch/fki/databases/iam-graph-database`

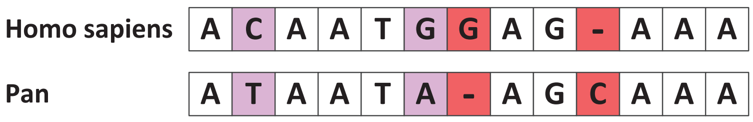


Figure 1.2: A simple alignment example on DNA sequences

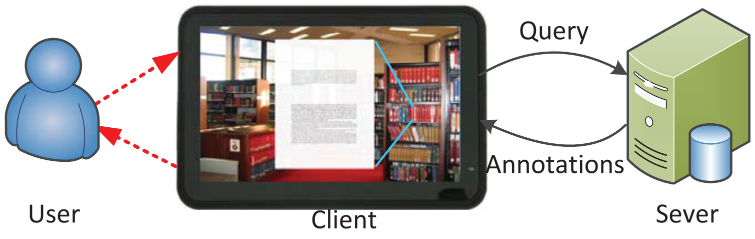


Figure 1.3: An example of the ebook annotation search

Now consider another example. Due to the fast development of the Internet, the number of public documents increases so rapidly that various copy detection techniques are proposed to protect the author's copyright. Among these techniques, string matching algorithms play important roles. Such as in [43], they have developed a match detect retrieval system using such algorithms.

In an ebook social annotation system, a large number of paragraphs are annotated and associated with comments and discussions[3]. For users who own a physical copy of the book, it is a very interesting feature to allow them to retrieve these annotations into their mobile devices using query by snapping. As shown in Figure 1.3, queries are generated by users when they use mobile devices to snap a photo of page in a physical book. The query photo is then processed by an optical character recognition (OCR) program which extracts the text from the photo as a sequence. Since the OCR program might generate errors within the sequence, we need to perform an approximate query against the paragraphs in the server to retrieve those paragraphs that had been annotated.

Obviously, most of the above interesting problems often require the similarity search of extremely long sequences. Although exiting approaches are effective on short sequence searches, they are less effective if there is a need to process sequences that are longer like a page of text in a book. This dissertation further investigates the long sequence similarity search problem from the viewpoint of enhancing efficiency, and focuses on the KNN sequence search problem as its more general usage in real applications.

[3]http://readpeer.com

1.1.3 Tree Structure: A Specific Case of Graph

In modern database applications, tree structure has been widely used to model the structured and semi-structured data. Typical examples of such data include RNA secondary structures [53, 77], XML documents [72], etc. An example of modeling a RNA secondary structure as a tree can be seen in Figure 1.4. Manipulating these tree structured data based on similarity also becomes essential for many applications. Consider the example on RNA secondary structure. Comparisons among the secondary structures are necessary to understanding the comparative functionality of different RNAs. This is because different RNA sequences can produce similar tree structures [53, 77]. In this case, algorithms to compute similarity measure between two trees are required.

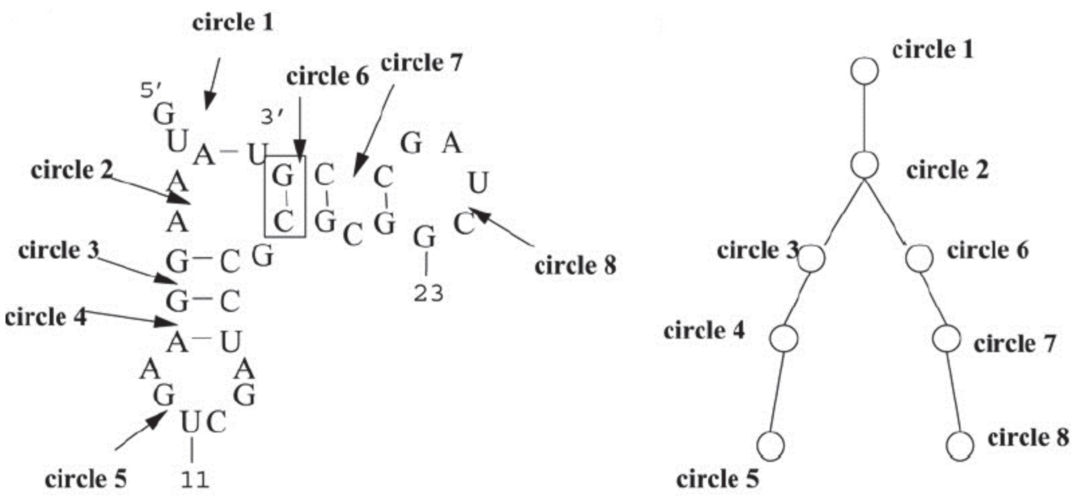


Figure 1.4: A tree model for a RNA secondary structure

Many existing works have studied the similarity measure and similarity search on large trees in huge databases (e.g., [37, 72]). In this dissertation, we see tree structure as a specific case of graph, and adapt the inverted index proposed in [?] to support the storage of tree data in our 3-in-1 unified system.

1.1.4 Complex and Nested Structures

In the real world, complex objects are not always restrictively modeled as single complex structures like sequences, trees and graphs. This study gives a new definition of nested structure where basic complex structures will be used as building blocks to construct more complex and nested structures.

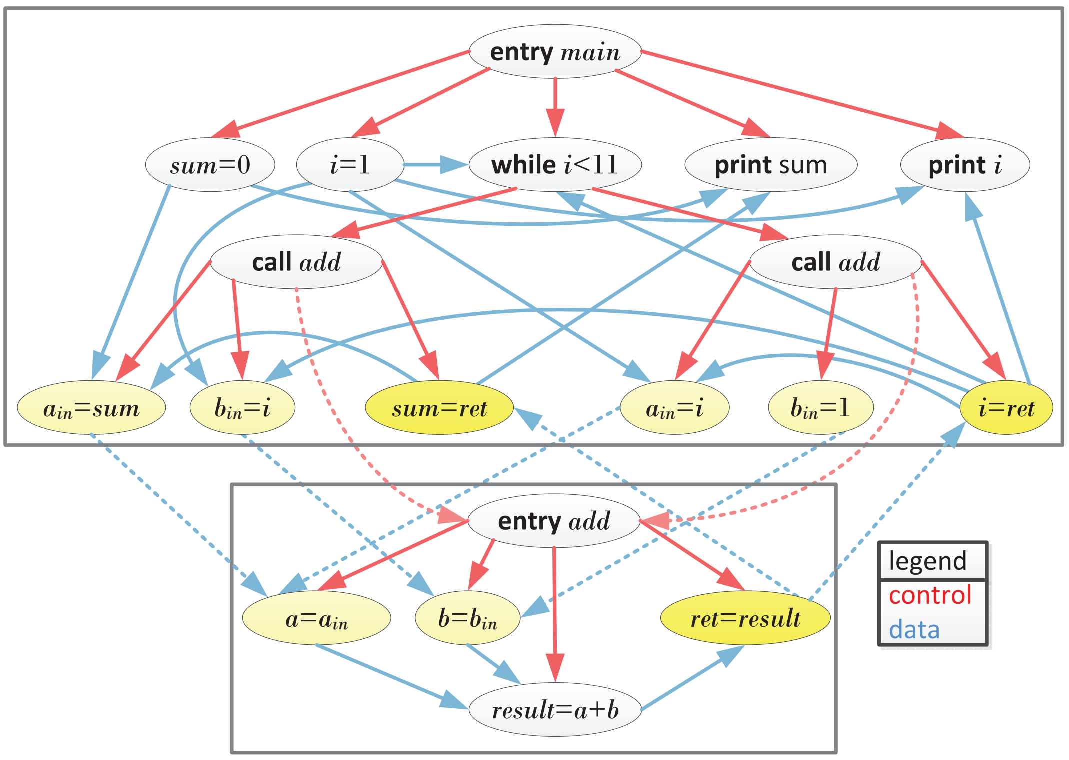


Figure 1.5: A nested program dependency graph

For example, in a generated dependency graph from program procedures, the relationship between procedures can then be represented by creating a higher level graph that connects the lower level dependency graphs. Figure 1.5 shows an example of such a simple nested graph with some vertices of program statements and two specific vertices of lower level dependency graphs. The nested structure poses new challenging research problems that do not exist in existing complex structure systems. How to search the required and interesting nested structures is a very important problem.

This study is also motivated by the real application on ebook social annotation systems. In our systems, an important application requires to identify ebooks with duplicate copies from different users for annotation sharing and recommendation. For example, users with similar interests prefer to upload the same ebook and an ebook can have multiple editions. This produces many duplicate copies of an ebook. Consequently, the need arises to support efficient document retrieval. Previous work models a web document as a simple graph [52]. Although the simple graph model is useful in the document classification tasks, it is not efficient to support the document retrieval task in our systems.

In particular, for an ebook with multiple editions, different editions may have different graph representations. In this dissertation, we use the nested structure to model an ebook document. For instance, a typical document might contain a title, authors, an abstract, and section headings. In this case, we can 1) use sequences to represent the title, author names, the abstract, and section headings; 2) convert each section heading into a vertex in the resulting document graph; 3) add an edge from a preceding section heading to a succeeding section heading. Therefore, a document is modeled as a nested graph with vertices of sequences. Figure 1.6 shows an example of such a simple nested graph with vertices of section heading sequences. With the nested graph representation, similarity search on vertex sequences is necessary first to generate candidates for further graph matching.

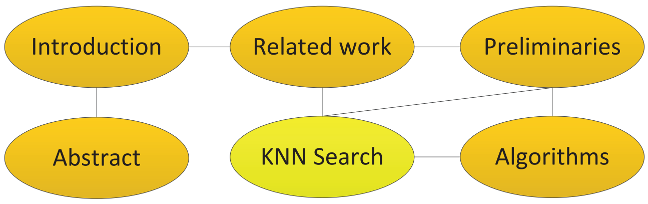


Figure 1.6: A nested document graph

It can be seen that all the above wide spectrum of application domains require proper storage and manipulation of complex and nested structures. This motivates to develop a general 3-in-1 indexing mechanism to support the efficient index and retrieval of complex structures.

1.2 Similarity Search on Complex Structures

Previous examples illustrate that similarity search on complex structures is very important in many applications. Enormous efforts have been put into developing practical searching methods on complex structures. Given a database of sequences, trees, or graphs, existing approaches attempt to find the most similar objects to a query object. No matter which type of complex structures is processed, the problems solved in most existing works can be categorized into four groups:

1. Full search: find complex structures that are identical to the query structure;

2. Substructure search: find complex structures that contain the query structure, or vice versa;

3. Full similarity search: find complex structures that are similar to the query structure based on a predefined similarity measure;

4. Substructure similarity search: find complex structures that contain the query structure based on a predefined similarity measure, or vice versa.

The above four kinds of queries are very useful within their own applications. As an example, the first two query problems on graph data are often formulated as search problems for graph or subgraph isomorphism [10, 29, 70]. However, in practice, exact matching is often too restrictive, as real structured objects are often affected by noise. Therefore, similarity searching for complex structures has become a basic research problem.

In general, different applications have various meanings by "similarity". For example, there are many sequence similarity measures, such as hamming distance, overlap coefficient, edit distance, and so on[4]. Likewise, many similarity measures have been proposed to evaluate the similarity between two graphs such as maximum common subgraph and graph edit distance. In the literature, edit distance (ED) has become a standard measure for various types of complex structures. In contrast to other measures, edit distance does not suffer restriction and can be applied to many applications. Consequently, most existing works concentrate on similarity search problems based on edit distance. In this dissertation, we generally formulate the problem of similarity search on complex structures as below.

Definition 1.1. ***Similarity search on complex structures***
Given a complex structure database $D = \{c_1, c_2, \ldots, c_n\}$ and a query structure q, find all the complex structures in D that are similar to q based on edit distance.

Hereafter, c_i is a sequence, a tree, or a graph. In general, users are interested in querying the complex structures within a specified tolerance based on edit distance. The edit distance on complex structures has been fully investigated in the literature [5, 20, 44]. Many existing works have proposed various definitions on *sequence edit distance* (SED), *tree edit distance* (TED), and *graph*

[4] http://en.wikipedia.org/wiki/Category:String_similarity_measures

edit distance (GED). This dissertation considers all these definitions and gives a general definition in edit distance on complex structures.

Definition 1.2. ***Edit distance on complex structures***
Given two complex structures c_1 and c_2, the edit distance between them, denoted by $\lambda(c_1, c_2)$, is defined as the cost of the least expensive sequence of edit operations that can transform c_1 to c_2. An edit operation can be an insertion, a deletion, or a substitution.

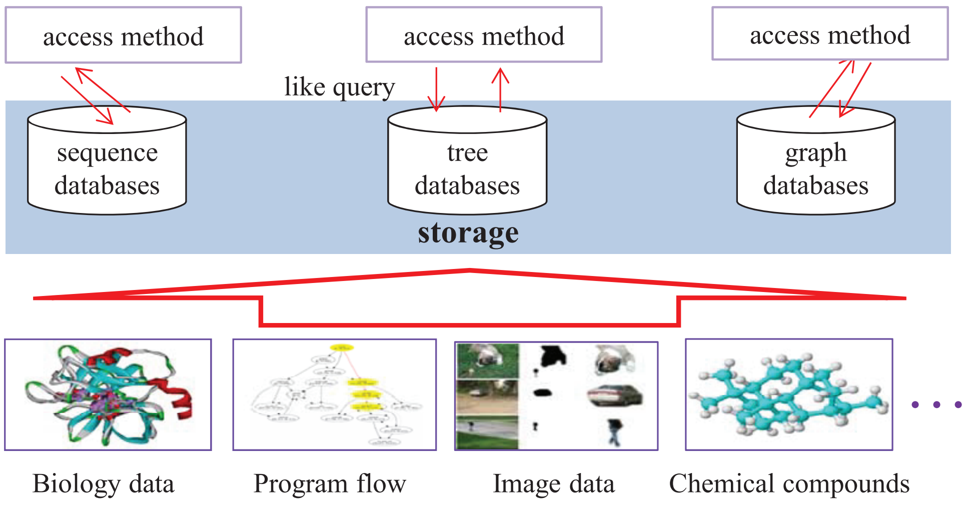


Figure 1.7: Existing systems for searching complex structures

As shown in Figure 1.7, existing works have been done on processing complex structures with isolated efforts targeted at specific domains. Although such works have focused on proposing efficient complex structure searching algorithms, they still suffer from certain drawbacks.

1. To support similarity search on graph databases, existing work follows a filter-and-refine framework. Based on filtering techniques, complex graph similarity computation can be reduced to enhance the graph search. Unfortunately, these methods have limitations. Some of them require enumerating sub-units exhaustively with high space and time overhead, and some of them do not capture the attributes on vertices or edges which are continuous values on graphs and often suffer from poor pruning power.

2. In sequence databases, the filter-and-refine framework works well in supporting the similarity search based on a signature-based schema and in-

verted files. However, these techniques are often constrained for answering similarity search on short sequences within a small distance threshold, and have been shown to have poor performance in KNN search.

3. Existing works typically consider complex structures separately for different applications. As shown in Figure 1.7, this results in wasting resources for data storage and requiring high cost when a real system requires to support the storage and retrieval of various types of complex structures. Especially for those real systems with complex and nested structures, to the best of our knowledge, no solution has been proposed.

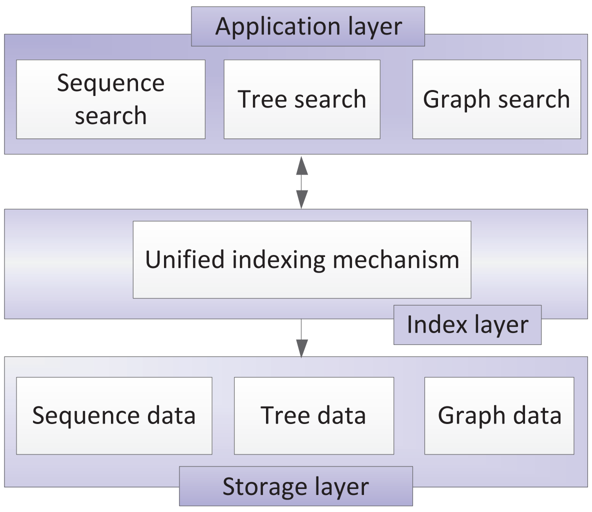


Figure 1.8: The 3-in-1 system architecture

To overcome the drawbacks, this study was to develop a unified framework, denoted by 3-in-1, that could support the efficient storage and retrieval of complex structures. Figure 1.8 shows our system architecture. The 3-in-1 system includes three layers: the storage layer, the index layer, and the application layer. To implement such system, the most important work was to design a unified indexing mechanism for supporting various complex structure search. Consequently, this dissertation focuses on addressing similarity search problems on complex structures using an inverted indexing structure. Therefore, the work of this research was to:

- propose a novel effective inverted indexing method for handling efficient graph similarity search.

- extend the novel index developed for graph similarity search to support efficient sequence similarity search, based on a novel pipeline framework.
- investigate the properties of complex and nested structures based on graph model and sequence model, and develop a unified 3-in-1 inverted index framework for various complex or nested structures.

The proposed 3-in-1 system may be useful for supporting different complex structures. A unified indexing mechanism could provide a general interface for various complex structures without redesigning their storage and retrieval. Moreover, the developed system should open up new applications that involve the model and search for a variety of complex or nested structures.

1.3 Summary of Contributions

In this dissertation, we seek to achieve the objectives described above on developing a unified 3-in-1 system that can support efficient storage and retrieval of various complex structures. The main contributions are summarized as follows:

- Our first contribution is to develop an efficient indexing mechanism for graph similarity search. We propose SEGOS, an indexing and query processing framework for graph similarity search. First, an effective two-level index is constructed off-line based on the star decomposition of graphs. Then, a novel search strategy based on the index is proposed. Two algorithms adapted from TA and CA methods [15, 57] are seamlessly integrated into the proposed strategy to enhance graph search. More specially, the proposed framework is easy to be pipelined to support continuous graph pruning. Extensive experiments that are conducted on two real datasets show the effectiveness and scalability of our approaches.
- Our second contribution is to further extend the index developed for graph similarity search to support efficient sequence similarity search. We focus on the problem of finding KNN results in sequence databases due to its more general usage in real applications. Unlike most existing works which used short, exact n-gram matching in a filter-and-refine framework, our approach allows us to use longer but approximate n-gram matching as a basis for pruning off KNN candidates. Based on this breakthrough,

we adopted a pipeline framework over a two-level index for searching KNN in the sequence database. By coupling this framework together with several efficient filtering strategies including the frequency queue and the well-known Combined Algorithm (CA), our proposal brings various enticing advantages over existing works, including 1) huge reduction on false positive candidates which will incur the expensive cost of verification; 2) progressive result update and early termination; 3) easily parallelizable. The results of extensive experiments on three real datasets show that our framework is effective and efficient to support the KNN sequence search.

- Our third contribution is to develop a unified 3-in-1 system for supporting efficient storage and retrieval of various complex or nested structures. We introduce a new concept of nested structure to model the complex data which are not easy to be represented using the single complex structures. To support efficient processing of the nested structures, we design a generic processing framework based on the inverted index structure. The proposed framework is applied to support various complex or nested structures. The input query can be a graph, a sequence, a tree, or a nested structure. We design the interface for answering queries for various complex structures. We also present a demo to show the application of the proposed unified framework on a real ebook social reading system[5].

Our works on graph similarity search and sequence similarity search were previously published in [67] and [68]. The real system for ebook social reading is published in http://readpeer.com/.

1.4 Thesis Organization

According to our contributions to solve the three problems, the rest of the thesis is organized as follows:

In Chapter 2, we give a thorough literature review of complex structure similarity search techniques as well as similarity measures and similarity search algorithms. We also provide all the preliminary concepts and notations used in the dissertation. For the graph approximate matching problem, we list exiting works based on the similarity measure that they adopted. While for the

[5]http://readpeer.com/

sequence similarity search problem, we give a comprehensive survey based on the index mechanism that they employed. We also list several works on tree similarity search based on the filtering techniques they used. We give a simple review on the 3-in-1 unified indexing problem, and introduce its application in our practical ebook social reading system. We also present several existing ebook reading tools.

In Chapter 3, we address the problem of similarity search on graph databases. We aim to develop a novel inverted index to speed up the graph similarity search. A two-level inverted index is first constructed based on the star decomposition method, and preprocessed to maintain a global similarity order both for decomposed stars and original graphs. With this blessing property, graphs can be accessed in increasing dissimilarity, and any GED based lower or upper bound can be used as filtering features. Consequently, we propose a novel pipeline search framework. Two algorithms adapted from TA and CA are seamlessly integrated into the framework, and it is easy to pipeline the proposed framework to process continuous graph pruning.

In Chapter 4, we study the problem of k-nearest neighbor sequence search based on the edit distance. We propose a novel pipeline approach using approximate n-grams. The approach follows a filter-and-refine framework. In the filtering phase, we develop a novel filtering technique based on counting the number of approximate n-grams. We also design an efficient searching algorithm with the frequency queue and the CA strategy. The frequency queue supports our proposed filtering techniques by reducing the number of candidate verification. By using the summation of gram edit distances as the aggregation function, the CA based search has an optimal feature of early termination which helps to invoke the halting condition of the whole pipeline framework. Our proposed filtering strategies have significant performance on the KNN search, and the pipeline framework is easy to support parallelism strategies.

In Chapter 5, we address some real challenging problems that exist in our real ebook social reading system, such as the annotation search problem, the ebook copy detection problem, and so on. To solve these problems, we introduce a new concept of nested structure and develop a unified indexing and searching framework to support efficient complex and nested structure search. We also present our ebook reading system which provides a friendly and collaborative annotation tool for social users.

In Chapter 6, we conclude remarks and discuss possible future extensions of the current work.

CHAPTER 2

Literature Review

Many existing works have been done on processing complex structures in various domains. These are isolated efforts to target at specific complex structures, such as sequences, trees, and graphs. In subsequent sections, we first give an overview of related works on graph similarity search problem, sequence similarity search problem, and tree similarity search problem. After that, we study the 3-in-1 search problem, and present several existing ebook social reading tools.

2.1 Graph Similarity Search Problem

As mentioned in previous chapter, this dissertation focuses on the graph similarity search problem based on edit distance. Since graph edit distance is important for supporting the graph similarity search, we first give a review on this graph similarity measure. However, existing works on this problem also use other similarity measures. To present a more complete study, we discuss and category different graph similarity search algorithms based on various similarity measures they adopted. We also give a simple review on related works on the graph isomorphism problem which only support the graph exact matching.

2.1.1 Graph Edit Distance

To support graph search based on similarity, a number of similarity measures have been proposed in the literature (e.g., [7, 17, 39]). Among them, graph edit distance (GED) is the most widely used measure for evaluating graph similarity. The GED problem has been extensively studied in many previous works, and a detailed survey can be found in [20]. GED is widely defined as the minimum number of edit operations needed to transform one graph into another. An edit operation can be an insertion, a deletion or a substitution of a vertex/edge. Algorithms for computing the GED can be classified into two classes: exact and approximate algorithms.

Exact algorithms calculate the exact GED between two graphs. Many optimal error-correcting subgraph isomorphism algorithms have been proposed, and A^*-based algorithms [28] are the most widely used ones. However, since GED computation is in NP-hard [21], these algorithms have exponential complexity and are only feasible for small graphs [46].

To avoid expensive GED computations, approximate algorithms are developed to compute lower and upper bounds of GED for graph filtering. In early works [1] and [31], GED computation was formulated as a BLP problem. They respectively computed a lower bound and an upper bound of GED with time complexity of $O(n^7)$ and $O(n^3)$. A recent method proposed in [75] computed both lower and upper bounds in cubic time, by breaking graphs into multi-sets of sub-units, and applying a novel algorithm to bound GED for filtering. Obviously, they take polynomial time on GED bound computation, which can efficiently reduce the total GED computation time by early pruning. However, such algorithms suffer from the scalability problem. Specifically, a full scan of the whole database brings in poor scalability in databases with a large number of graphs. To solve this problem, it is natural to consider building an effective index structure to reduce GED computations for the graph similarity search.

2.1.2 Graph Isomorphism Search

In graph isomorphism and subgraph isomorphism search, the aim is to find graphs that are either isomorphic or contain a subgraph that is isomorphic to the query graph. In this regard, the matching must be exact and there is no query relaxation of any form. Algorithms for isomorphism search includes **FG-index** [10], **TreePi** [78] and **Tree+Delta** [80]. These methods differ only in the features that they use for pruning candidates. These techniques however cannot be easily generalized to handle graph similarity search which requires certain amount of error tolerance in the matching graphs.

2.1.3 Graph Similarity Search

There is a great amount of literatures on graph similarity search. However, few developed indexes for searching by graph edit distance. Here we list these works based on the similarity function that they adopted.

Feature Counting

Since graph alignment is NP-hard, various heuristical feature counting methods have been developed to compare graphs. **GraphGrep** proposed in [22] compares graphs by counting the number of matching paths between two graphs. Signatures are generated for all the paths in a graph up to a threshold length and inserted into an index to facilitate searching and counting of paths. In [58], features are generated by merging each node in a graph together with its neighbouring vertices information. Graph similarity is judged by counting the number of features that are sufficiently from both graphs and a B$^+$-tree is used to index the features of the graphs in the database. However, none of these methods can guarantee that edit distance is minimized for graphs returned as query results.

Edge Relaxation

Given two graphs g_1 and g_2, if c_{12} is the maximum common subgraph of g_1 and g_2, then the substructure similarity between g_1 and g_2 is defined by $\frac{|E(c_{12})|}{|E(g_2)|}$ and $1-\frac{|E(c_{12})|}{|E(g_2)|}$ is called the edge relaxation ratio. In [70], the **gIndex** is developed to support similarity search by edge relaxation. The **gIndex** adopts discriminative frequent subgraphs as basic indexing structures and involves complex feature extraction for each query. Adopting edge relaxation as a similarity measure implicitly excludes node substitution as a graph edit operation [75] and is thus not general enough to handle search by edit distance.

Edit Distance

As far as we know, there are few works that provide an index for searching by graph edit distance. The **C-Tree** [29] is one of such pieces of work. In **C-Tree**, an R-tree like index structure is used to organize graphs hierarchically in a tree. Each internal node in the tree summarizes its descendants by a graph closure. By approximating the graph edit distance against the graph closures that are stored in the internal nodes, **C-Tree** tries to avoid accessing individual graphs that are too dissimilar based on the GED. A most recent work κ-**AT** [63] decomposes graphs into κ-*adjacent tree* patterns and indexes them using inverted lists. A lower bound is also proposed to filter out graphs that do not sharing sufficient common patterns with a query graph.

In this dissertation, we focus on the graph similarity search problem based on edit distance. Although two state-of-the-art works, **C-Tree** [29] and κ-**AT** [63], have been made some progress on solving this problem, they still suffer several serious limitations. The κ-**AT** has been shown to be efficient on pruning using the inverted index. However, the GED bound they derived is so loose that it generates too many false positives which will incur the expensive cost of verification. The **C-Tree** takes more filtering time than the κ-**AT** to reduce the false positive candidates, which can save the verification cost. However, the filtering power of this method is still poorer than that of those works with tighter GED bounds [1, 31, 75]. As described in Section 2.1.1, there is no indexing technique that has been proposed to support the tighter GED bounds. This motivates our first work on graph similarity search to design such a novel indexing method. We will illustrate this work in Chapter 3.

2.2 Sequence Similarity Search Problem

Sequence similarity search based on edit distance is a well-studied problem (e.g., [39, 47, 69]). An extensive survey had been conducted very early in [44]. We first give a review on sequence edit distance, and then summarize exiting sequence similarity search algorithms by the various filtering techniques they have employed.

2.2.1 Sequence Edit Distance

To compute the exact sequence edit distance (SED), existing algorithms can be classified into three groups: dynamic programming, automata, and bit-parallelism [44]. Among them, dynamic programming algorithms are the most well-known algorithms for computing the exact SED. Given two sequences s_1 and s_2, the basic idea computes $\lambda(s_1, s_2)$ based on dynamic programming. A two-dimensional cost matrix $M_{0..|s_1|,0..|s_2|}$ is first used to hold edit distance values, where $M_{i,j}$ represents the best score to match $s_1[1,i]$ to $s_2[1,j]$[1]. It is computed as follows:

$$M_{i,j} = \min \begin{cases} M_{i-1,j-1} + \delta(s_1[i], s_2[j]) & \textit{substitute/copy} \\ M_{i-1,j} + \delta(s_1[i], \varepsilon) & \textit{insert} \\ M_{i,j-1} + \delta(\varepsilon, s_2[j]) & \textit{delete} \end{cases}$$

where δ is an arbitrary distance function on characters. Let $M_{0,0} = 0$, $M_{i,0} = i$ and $M_{0,j} = j$, representing distances between two sequences including empty sequence. A dynamic programming algorithm fills each cell of the matrix by computing its upper-left, upper, and left neighbors. It takes $O(|s_1||s_2|)$ time and $O(\min(|s_1|, |s_2|))$ space. Then we finally obtain $\lambda(s_1, s_2) = M_{|s_1|,|s_2|}$.

Many existing works focus on speeding up the dynamic programming computation, the most efficient algorithm requires $O(|s|^2/\log|s|)$ time [39] for computing the SED, and only $O(\tau|s|)$ time for testing if the SED is within some threshold τ [79].

2.2.2 Sequence Similarity Search

As described above, early similarity search algorithms are based on online sequential search, and mainly focus on speeding up the exact sequence edit distance (SED) computation using the above exact SED computation algorithms. However, these online algorithms still suffer from poor scalability in terms of sequence length or database size since they need a full scan on the whole database. To overcome this drawback, most recent works follow a filter-and-refine framework. Many indexing techniques have been proposed to prune off most of the sequences before verifying the exact edit distances for a small set of candidates [45]. There are three main indexing ideas: enumerating, backtracking and partitioning.

The first idea is introduced for supporting specific queries when strings are very short or the edit distance threshold is small (e.g., [3, 66]). It is clear that enumeration usually have high space complexity and is often impractical in real query systems.

The second idea is based on branch-and-bound techniques on tree index structures. In [9, 64], a trie is used to index all strings in a dictionary. With

[1]The position of the first character in a sequence is 1 instead of 0.

a trie, all shared prefixes in the dictionary are collapsed into a single path, so they can process them in the best order for computing the exact SEDs. Sub-trie pruning is employed to enhance the efficiency of computing the edit distance. However, building a trie for all strings is expensive in term of both time and space complexity. In [79], a B^+-tree index structure called B^{ed}-tree is proposed to support similarity queries based on edit distance. Although this index can be implemented on most modern database systems, it suffers from poor query performance since it has a very weak filtering power.

To improve filtering effectiveness, most existing works employ the third idea that splits original strings into several smaller signatures to reduce the approximate search problem to an exact signature match problem (e.g., [8, 25, 35, 36, 38, 48, 56, 61, 65, 73]). We further classify these methods based on their preprocessing methods into the *threshold-aware* approaches and the *threshold-free* approaches.

The *threshold-aware* approaches have been developed mainly based on the prefix-filtering framework. Recent work in [65] performed a detailed studies of these methods [38, 48, 65] and conclude that the prefix-filtering framework can be enhanced with an adaptive framework. These methods typically work well only for a fixed similarity threshold. If the threshold is not fixed, two choices exist. First, the index has to be built online for each query with a distinct threshold. This could be time consuming and always be impractical in real systems. Second, multiple indexes are constructed offline for all possible thresholds. This choice has high space complexity especially for databases with long sequences since there can be many distinct edit distance thresholds.

The *threshold-free* approaches generally employ various n-gram based signatures. The basic idea is that if two strings are similar they should share sufficient common signatures. Compared to the *threshold-aware* approaches, these methods generally have much less preprocessing time and space overhead for storing indexes. However, if we ignore the preprocessing phrase, these methods have been presented to have the worse performance for supporting edit distance similarity search [48]. This is because they often suffer from poor filtering effectiveness through the use of loose bounds.

Although such approaches may be efficient for approximate searching with a predefined threshold, limited progress has been made for addressing the KNN search problem. However, the KNN search problem has wider usage in practice.

2.2.3 KNN Sequence Search

To solve the KNN sequence search problem, existing efforts utilize two kinds of index mechanisms [14, 62, 74, 79].

The first index mechanism is adapted from inverted list based index [62, 74]. The KNN search algorithm employs the same intuition by selecting candidates with sufficient number of common n-grams. The difference between them is the list merging technique. In [62], the MergeSkip algorithm is employed to reduce the inverted list processing time. A predefined threshold based algorithm is also proposed by repeating the approximate string queries multiple times to support KNN search. In [74], the basic length filtering is used to improve the inverted list processing.

Another index mechanism is based on the tree structure [14, 79]. In [79], a B^+-tree based index is proposed to index database sequences based on some sequence orders. The tree nodes are iteratively traversed to update the lower bound of edit distance and the nodes beyond the bound are pruned. In the most recent work [14], an in-memory trie structure is used to index sequences and share computations on common prefixes of sequences. A range-based method is proposed by grouping the pivotal entries to avoid duplicated computations in the dynamic programming matrix when the edit distance is computed. Although such approaches are effective on the short sequence search, their performances degrade for long sequences since the length of the common prefix is relatively short for long sequences and the large number of long, single branches in the trie brings about large space and computation overhead.

To overcome the drawbacks of the above existing work, this dissertation proposes the second work which attempts to derive tighter SED bounds and extend the inverted index proposed for the first work to enhance the sequence search. The detail of this work will be presented in Chapter 4.

2.3 Tree Similarity Search Problem

Exiting works on the tree similarity search problem have focused on proposing efficient indexing techniques and filtering algorithms. As this dissertation sees tree as a specific case of graph, we present a simple overview on the related works which help to build our final unified 3-in-1 system.

To compute the exact tree edit distance (TED), numerous algorithms are proposed in the literature, and a complete survey can be found in [5]. Computing TED has been shown to be in NP-complete in previous works [5]. Although several works have introduced the concept of constrained edit distance and proposed polynomial algorithms, due to the high computational complexity, it is still impractical to directly use TED for searching huge tree databases. Consequently, previous efforts are often put into finding efficient filtering methods.

Most existing searching methods follow a filter-and-refine framework. They aims to find efficient and tight bounds to guarantee the filtration efficiency. In general, two main ideas are used: transforming complex trees into simple sequences by using SED to bound the TED (e.g.,[26, 37]), and adapting q-gram methods by breaking trees into smaller sub-units (e.g., [72]). The *sequence-based approach* first transforms original trees into their corresponding preorder and postorder traversal sequences. Then the SED of two sequences is used as the bound of the TED. Pairs of trees from heterogeneous repositories are matched when their SEDs are within a threshold. As previous review on SED, the quadratic time of SED computation is also not so efficient for every pair comparison in the whole database. Differently, the *q-gram like approach* breaks trees into a set of smaller sub-units (like binary branches in [72]). Based on storing these sub-units using inverted index, trees are mapped into an approximate numerical multidimensional vectors which encodes the original structure information and distance of vectors are used as a lower bound of TED. This index mechanism has been shown to be effective on supporting the tree similarity search [72]. In this dissertation, we see tree structure as a specific case of graph, and adapt the inverted index proposed in [72] to support the storage of tree data in our 3-in-1 unified system.

2.4 3-in-1 Unified Indexing Problem

Existing systems process various types of complex structures with isolated efforts, targeting at specific domains. Yan et al. [27] investigated the importance of mining and searching problems in complex structures like graphs, trees, and networks. However, they still cater to the storage of complex structures separately. This results in a waste of resources for redesigning the index mechanism and developing numerous query processing algorithms for each specific application. This dissertation aims to develop a unified storage system.

Based on the above literature review, we observe that the storage method based on inverted lists can be used to solve the similarity search problems on various types of complex structures. We summarize the idea of such approaches as "shotgun and assembly". The idea is that complex structures will be first broken down into smaller units, such as q-grams for sequences (e.g., [8, 35]), binary branches for trees (e.g., [72]), and stars for graphs (e.g., [75]). Then, smaller units are stored in inverted lists with each inverted list keeping track of references from complex structures to the corresponding smaller unit. Similarity search on such complex structures can be effectively performed by breaking them down into smaller units, after which searches are performed by retrieving these smaller units individually in the inverted lists and assembling them. Consequently, this dissertation propose the third work to adopt this idea to design a unified 3-in-1 inverted index storage for various complex structures.

2.4.1 The Storage of Inverted Index

Many existing works have focused on proposing an appropriate storage schema for creating and managing inverted files. Such approaches are mainly developed to support efficient information retrieval, and a earlier comprehensive survey can be found in [82].

Several works directly used the file systems to store and manage inverted files. A most recent work [4] has designed a disk-based method to support efficient sequence similarity search. In such approaches, the most challenging problem would be the cost of update. As inverted lists are stored in sequences of blocks, the focus on reducing update costs thus may lead to increased space consumption and slower query evaluation. Considering this problem, [76] has observed that inverted indexes can also be implemented in commercial relational database systems.

Many works used the relational database management system (RDBMS) (e.g., [6, 12, 13, 16, 23, 41, 49, 54, 59]) to manage inverted files. In these works, two main conventional storage structures are used. We call them as the *table-based approach* and the *tree-based approach*. The *table-based approach* [6, 23, 54] uses a persistent object store to manage inverted files. That is to store a table of records consisting of a keyword and a posting in a database.

Such approaches can simplifies implementation and use intelligent caching or contiguous storage to improve information retrieval. However, they still suffer low query performance and require excessive storage space due to redundancy of keywords. Differently, the *tree-based approach* [12, 13, 16, 41, 49, 59] uses tree structures instead of database tables for storing the inverted index. Such approaches has focused on various important issues such as index compression, incremental updates and distributed query performance. Especially, this approach is also adopted in [24] where n-grams are stored in a relational database to support approximate string join. This dissertation will further investigate this problem in Chapter 5.

2.4.2 Social Reading Tools

As mentioned in Chapter 1, the unified index mechanism helps to solve many real challenging problems in social reading systems. Here, we present a review on existing ebook reading tools.

The development of digital publishing provides new possibilities for users to share their ideas and connect to each other [40]. Early e-book readers support several simple features, such as permitting a user to highlight text, write sticky notes, and track annotations. For example, Sony Reader is introduced in 2006, which sets the standard for eInk devices[2]. With such devices, users can only track previous annotations without any feedback by commenting back. The need arises to provide an information sharing tool for users to leave their comments and start conversations with other users. Some later reading systems have been developed to allow users to share their readings and discuss books, such as Goodreads[3] and Shelfari[4]. Goodreads allows users to create short book reviews and share comments with their friends; while Shelfari focuses on exciting users to find those users with common reading interests. However, such sites show the comments of users separately from original books. Differently from e-book readers, such sites do not allow users to see the book contents.

Consequently, recent social reading systems focus on developing a user-friendly platform by combing the features of highlighting with the capability of social networking. Since 2008, more and more reading sites have been pub-

[2]http://en.wikipedia.org/wiki/Sony_Reader
[3]http://www.goodreads.com
[4]http://www.shelfari.com

lished, such as BookGlutton[5], Readmill[6], ReadSocial[7], and so on. Although such systems have certain successful features by providing users various services, limited progress has been made for reading data management and retrieval. In particular, with the development of web browser plugins and mobile applications, a cross-system information management tool is required. ReadSocial has offered an interface for users to create virtual group on top of different reading systems, and users can share their comments anywhere in any group by group tags. However, such systems require users to be very familiar with group tags or contents. Otherwise, it may require quality group recommendation, and challenging technical considerations on reading recognition. To solve such challenging problems, a most recent social reading tool requires novel techniques. In this dissertation, we will illustrate how to employ the unified storage system for efficiently managing information in such social reading systems. The demo system for social reading will be presented in Chapter 5.

[5] http://www.bookglutton.com
[6] https://readmill.com
[7] https://www.readsocial.net

CHAPTER 3

An Efficient Graph Indexing Method

Since graph is the most general model and graph similarity search is the most hard and challenging problem, this work first addresses a novel efficient indexing method for handling the graph similarity search. In this work, a two-level inverted index is introduced together with a set of effective and efficient pruning techniques. Comprehensive experiments on two real datasets also show that the proposed method outperforms the state-of-the-art works.

3.1 Overview

As mentioned in Chapter 1, graphs are widely used to model complex entities in many applications, and managing a large amount of graph data is a very challenging problem. It is essential to process graph queries efficiently. The classical query processing is often formulated as the (sub)graph isomorphism problem. However, this kind of exact matching is too restrictive, as real objects are often affected by noises. Therefore, similarity search has become a basic operation in graph databases.

This work has focused on the graph similarity search problem based on edit distance. This problem can be described as follows: *given a graph database $D = \{g_1, g_2, \ldots, g_{|D|}\}$ and a query graph q, find all $g_i \in D$ that are similar to q within a GED threshold denoted by τ.* Scanning the whole database D to compute the GED between q and each $g_i \in D$ is very expensive, due to the high complexity of GED computation, which is proved to be in NP-hard. Facing this difficulty, several existing works use upper and lower bounds of GED to prune off unlikely candidates. Although these methods allow more efficient bound computations, they still suffer from certain drawbacks. First, GED bound computations are still very expensive. Second, they do not take full advantage of indexes, and require a full scan of the whole database. These bring in poor scalability in databases with a large number of graphs.

Facing these difficulties, it is natural to consider building an effective index structure to reduce complex computations. Our basic idea is to break graphs into sub-units (sub-unit is used as a small substructure derived from a graph in this work), and to index them as filtering features using inverted lists. In our approach, we decompose each database graph into sub-units, and each sub-unit contains a vertex and discriminative information about its neighboring vertices and edges. To avoid exhaustive enumerations, discriminative information for a sub-unit only contains the most neighboring information. To enhance filtering power, the decomposed sub-units in our method are compared against the sub-units generated from the query graph using the Hungarian algorithm. Formulated as a bipartite matching problem, each sub-unit in database graphs can have only partial matching with each sub-unit in the query graph. The need arises to find highly similar sub-units that not only match exactly but also are similar to the sub-units from the query.

To support such functionality, we propose a novel query processing framework, called **SEGOS** (**SEarching similar Graphs based On Sub-units**). In this framework, a two-level inverted index is constructed based on the decomposed sub-units. In the upper-level index, sub-units derived from the graph database are used to index all graphs using inverted lists. In the lower-level index, each sub-unit is further broken into multiple vertices and indexed in inverted lists. This two-level inverted index is preprocessed to maintain a global order for sub-units and graphs. This order ensures that sub-units or graphs can be accessed in increasing dissimilarity to a query sub-unit or graph. Given a query, our strategy follows a novel, cascaded framework: in the lower level, top-k similar sub-units to each sub-unit of the query can be returned quickly; in the upper level, graph pruning is done based on the top-k results from the lower level. Two search algorithms, based on the paradigm of the TA and the CA methods are proposed for retrieving sub-units and graphs. By deploying the summation of sub-unit distances as the aggregation function, sorted lists can be easily constructed to guarantee the global orders on increasing dissimilarity for graphs. The CA based methods can enhance similarity search by avoiding access to graphs with high dissimilarity. It is clear that the top-k sub-units returned from the lower-level sub-unit search can be automatically used as the input to the upper-level graph search. Therefore, these two search stages are easy to be pipelined to support continuous graph pruning.

In summary, the main contributions of this work are:

- We propose a novel two-level inverted index to speed up graph similarity search. The lower-level index is first used to efficiently find top-k similar sub-units. With the top-k results, the upper-level index is retrieved to construct a list of graphs that are sorted based on the similarity score.
- We propose a better search strategy following a cascade framework using the novel index. Search algorithms adapted from the TA and the CA methods [15] are proposed to improve efficiency by dramatically reducing accesses to sub-units and graphs with high dissimilarity.
- **SEGOS** can be applied to enhance existing works like **C-Star** [75] developed for evaluating graph edit distance using sub-units.
- **SEGOS** is easy to be pipelined into three processing stages: the lower-level top-k sub-unit search, the upper-level graph sorted list processing, and the dynamic graph mapping distance computation.

3.2 Indexing and Filtering Techniques

In this work, we focus on a database D of undirected, simple graphs whose vertices are labelled. A graph is defined as a 4-tuple $g = (V, E, \Sigma, l)$, where V is a finite set of vertices, $E \subseteq V \times V$ is a set of edges, Σ is a finite alphabet of vertex labels and $l : V \to \Sigma$ is a labelling function assigning a label to a vertex. Figure 3.1 shows an example of a graph database with five data graphs from g_1 to g_5. The size of a graph g, denoted by $|g|$, is the number of vertices in g, and other common notations used in this work can be found in the "LIST OF SYMBOLS" in Page xii.

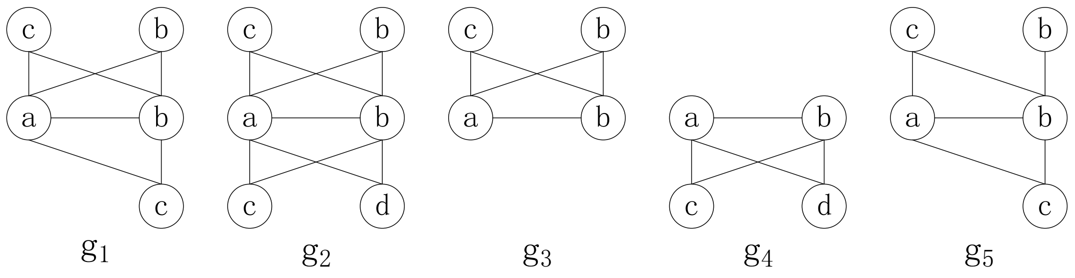


Figure 3.1: A sample graph database

3.2.1 Graph Decomposing Method

To estimate GED bounds effectively, we employ the idea proposed in [75] to decompose a graph into multiple sub-units like star. A star is defined as a labelled, single-level and rooted tree which can be represented by a 3-tuple $st = (r, L, l)$, where r is the root, L is the set of leaves and l is a labelling function. For each v_i in the graph, we construct a star $st_i = (v_i, L_i, l)$, where L_i is the label set of v_i's neighbors. A graph g with $|g|$ vertices can be decomposed into a multiset of $|g|$ stars. In Figure 3.2, two graphs g_1 and g_2 are transformed into two star representations: $S(g_1)$ and $S(g_2)$. With this transformation, we cite a lemma given in [75] to compute the edit distance between two stars.

Lemma 3.1. *(STar Edit Distance)(STED) Given two stars st_1 and st_2, the edit distance between them is computed as*

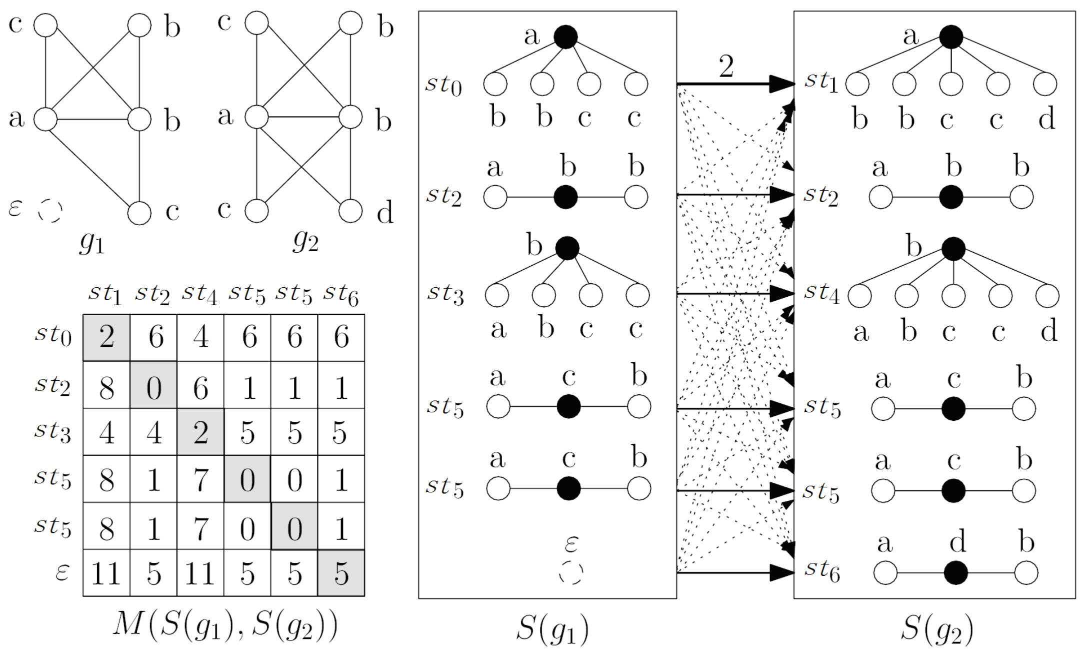
$$\lambda(st_1, st_2) = T(r_1, r_2) + d(L_1, L_2)$$

where $T(r_1, r_2) = 0$ if $l(r_1) = l(r_2)$, otherwise $T(r_1, r_2) = 1$.

$$d(L_1, L_2) = \big||L_1| - |L_2|\big| + M(L_1, L_2)$$

$$M(L_1, L_2) = \max\{|\Psi_{L_1}|, |\Psi_{L_2}|\} - |\Psi_{L_1} \cap \Psi_{L_2}|$$

Ψ_L is the multiset of vertex labels in L. Assuming that the alphabet Σ of vertex labels has a total order, we can compute STED between two stars in only $\Theta(n)$ time, if Ψ_{L_1} and Ψ_{L_2} are sorted. For example, to compute the distance between st_0 of $S(g_1)$ and st_1 of $S(g_2)$ in Figure 3.2, it is obvious that $T(r_1, r_2) = 0$, for $l(r_1) = l(r_2) = a$. Having $|L_1| = 4$, $\Psi_{L_1} = \{b, b, c, c\}$, $|L_2| = 5$, and $\Psi_{L_2} = \{b, b, c, c, d\}$, we can compute the STED as $\lambda(st_0, st_1) = 0 + |4 - 5| + 5 - 4 = 2$.



$$
\begin{array}{c|cccccc}
 & st_1 & st_2 & st_4 & st_5 & st_5 & st_6 \\
\hline
st_0 & \boxed{2} & 6 & 4 & 6 & 6 & 6 \\
st_2 & 8 & \boxed{0} & 6 & 1 & 1 & 1 \\
st_3 & 4 & 4 & \boxed{2} & 5 & 5 & 5 \\
st_5 & 8 & 1 & 7 & \boxed{0} & 0 & 1 \\
st_5 & 8 & 1 & 7 & 0 & \boxed{0} & 1 \\
\varepsilon & 11 & 5 & 11 & 5 & 5 & \boxed{5}
\end{array}
$$

$M(S(g_1), S(g_2))$

Figure 3.2: Mapping distance computation between g_1 and g_2

Definition 3.1. *(Mapping Distance) Given two star representations $S(g_1)$ and $S(g_2)$ with the same cardinality, assume $P : S(g_1) \to S(g_2)$ is a bijection, then the distance between them is defined as*

$$\mu(g_1, g_2) = \min_P \sum_{st_i \in S(g_1)} \lambda(st_i, P(st_i))$$

The computation of mapping distance is equivalent to finding an optimal mapping between two star representations. Zeng *et al.* [75] constructs a weighted matrix for each pair of stars from two graphs, and applies the Hungarian algorithm [33] to get the optimal solution in cubic time. The weight between two stars is the STED. If two graphs are of different size, ϵ node is inserted for normalization. In Figure 3.2, the bottom left matrix $M(S(g_1), S(g_2))$ is the weight matrix between star sets $S(g_1)$ and $S(g_2)$. Cells in gray denote the optimal matching between $S(g_1)$ and $S(g_2)$, i.e. $\mu(g_1, g_2) = 2+0+2+0+0+5 = 9$. To have a clear view, two sets of stars are shown, and the optimal matching is marked with solid arrows.

[75] shows that the mapping distance can be used to bound GED effectively, and a lower bound $L_m(g_1, g_2)$ and a upper bound $U_m(g_1, g_2)$ can be derived as below.

Lemma 3.2. *Suppose $\mu(g_2, g_1)$ is the mapping distance between g_1 and g_2. Then,*

$$L_m(g_1, g_2) = \frac{\mu(g_2, g_1)}{\max\{4, [\max\{\delta(g_1), \delta(g_2)\} + 1]\}} \leq \lambda(g_1, g_2)$$

Lemma 3.3. *Suppose P is a mapping between $V(g_1)$ and $V(g_2)$ obtained from Hungarian algorithm when computing $\mu(g_1, g_2)$. Then $U_m(g_1, g_2) = C(g_1, g_2, P)$ $\geq \lambda(g_1, g_2)$, where $C(g_1, g_2, P)$ is the cost to transform g_1 to g_2 with P [31].*

This work employs the above decomposing method to build the index, hereafter, a sub-unit refers to a star structure, and STED can also denote the sub-unit edit distance. The sub-unit is also represented as a sequence of labels for simplicity. For example, in Figure 3.5, "st_0: abbcc" represents the sub-unit st_0 as its label sequence of "abbcc". As shown above, computing mapping distance takes cubic time on graph size. The existing filtering strategy proposed in [75] suffers from poor scalability as it has to scan a large graph database, and compute mapping distance between each data graph and the query graph for pruning. Facing this problem, two ways can be developed to enhance the graph search: using dynamic mapping distance computation and a better filtering strategy.

3.2.2 Dynamic Mapping Distance Computation

To reduce complex mapping distance computations, this work proposes a novel computing method as below.

Theorem 3.1. *Given two graphs g_1 and g_2 and their sub-unit representations $S(g_1)$ and $S(g_2)$. Suppose $S'(g_2)$ contains several sub-units derived from g_2 and $S'(g_2) \subseteq S(g_2)$. Then we have*

$$\mu(S(g_1), S'(g_2)) \leq \mu(g_1, g_2)$$

In Figure 3.3, $M(S(g_1), S'(g_2))$ is a different cost matrix defined for computing $\mu(S(g_1), S'(g_2))$. For the ϵ sub-unit, we define its distance to any existing sub-unit st_i in $S(g_1)$ as 0 instead of $\lambda(st_i, \epsilon)$. We apply the Dynamic Hungarian [60] to find the minimum cost and matching on $M(S(g_1), S'(g_2))$. After that, the incremental part for computing full $\mu(g_1, g_2)$ uses the original definition of cost matrix with $\lambda(st_i, \epsilon)$. With this definition, it is clear that $\mu(S(g_1), S'(g_2)) \leq \mu(S(g_1), S(g_2))$.

$$\begin{array}{c|c|c|c|c|c|c|}
 & \varepsilon & st_2 & \varepsilon & st_5 & st_5 & \\ \hline
st_0 & \cancel{0} & 6 & 0 & 6 & 6 & \\ \hline
st_2 & 0 & 0 & 0 & 1 & 1 & \\ \hline
st_3 & 0 & 4 & \cancel{0} & 5 & 5 & \\ \hline
st_5 & 0 & 1 & 0 & 0 & 0 & \\ \hline
st_5 & 0 & 1 & 0 & 0 & 0 & \\ \hline
 & & & & & & \\ \hline
\end{array}
\qquad
\begin{array}{c|c|c|c|c|c|c|}
 & st_1 & st_2 & st_4 & st_5 & st_5 & st_6 \\ \hline
st_0 & 2 & 6 & 4 & 6 & 6 & 6 \\ \hline
st_2 & 8 & 0 & 6 & 1 & 1 & 1 \\ \hline
st_3 & 4 & 4 & 2 & 5 & 5 & 5 \\ \hline
st_5 & 8 & 1 & 7 & 0 & 0 & 1 \\ \hline
st_5 & 8 & 1 & 7 & 0 & 0 & 1 \\ \hline
\varepsilon & 11 & 5 & 11 & 5 & 5 & 5 \\ \hline
\end{array}$$

$$M(S(g_1), S'(g_2)) \qquad\qquad M(S(g_1), S(g_2))$$

Figure 3.3: An example for computing $\mu(S(g_1), S'(g_2))$

This property allows us to compute bounds for the GED between two graphs even if only a subset of a graph's sub-units are available. If $\mu(S(q), S'(g))$ is sufficiently large, there is no need to compute the bound based on the full set of sub-units between graphs.

3.2.3 CA-based Filtering Strategy

To reduce the complex computations of GED bounds, it is natural for us to consider a more efficient filtering strategy. In this work, we propose a novel search strategy based on the paradigm of the TA and the CA methods proposed in [15]. As far as we know, such TA and CA based methods had never been previously applied for matching complex structures like sequences (using qgrams) [36], trees (using binary branches) [?], or graphs [22, 29, 70, 78, 10, 63]. This is because all these previous methods simply use the number of exact matches among the sub-units to bound the edit distance and compute the exact edit distance for all candidate that pass through the filter. For cases in which such filters are not effective (eg. range query with a very loose edit distance threshold), our approach here provide an elegant way to avoid computing the exact edit distance for large number of candidates.

Figure 3.4 shows a simple example that helps to illustrate our CA-based filtering strategy for range query on the graph database in Figure 3.1. Consider the three score sorted lists on the left which consist of sub-units from q. Each entry in the lists records the graph identity g_i and the STED between the corresponding sub-unit in g_i and the sub-unit of q. We use the summation of STEDs as the score aggregation function and assume that for an unseen graph

$q : st_0$	
gid	λ_1
g_1	0
g_2	3
g_3	3
g_4	3
...	...

$q : st_1$	
gid	λ_2
g_2	0
g_1	2
g_3	2
g_5	2
...	...

$q : st_2$	
gid	λ_3
g_3	0
g_1	0
g_2	1
g_4	1
...	...

$\omega = \lambda_1 + \lambda_2 + \lambda_3$			
ω		gid	μ
0		g_1	2
3		g_2	4
5		g_3	6
$\tau * \delta' = 1 * 4 = 4$ Halt: $\omega = 5 > 4$			

Figure 3.4: A simple example for CA-based filtering strategy

g, $\mu(g, q) \geq \omega$ where ω is the summation of STEDs seen currently (we also call this assumption as *monotonic assumption*). Then, in this example, the search algorithm halts when $\omega = \lambda_1 + \lambda_2 + \lambda_3 = 5 > \tau * \delta'(= 4)$. Hereafter, we denote $\delta' = \max\{4, [\max\{\delta(q), \delta(D')\} + 1]\}$ where D' is the set containing all unseen graphs. Here, g_4 and g_5 are filtered out without computing their mapping distances, since their values of μ are no less than ω. From Lemma 3.2, for an unseen graph g with $\mu(g, q) > \tau * \delta'$, we have $\lambda(g, q) > \tau$ and g can be safely filtered out.

Accordingly, our search strategy must overcome the following challenges: 1) An effective indexing structure is needed for constructing the score-sorted lists. 2) Since graphs in score indexing lists are sorted according to their STEDs to the sub-unit of the query, an efficient search algorithm must be developed to obtain sub-units that are highly similar to the query sub-unit. 3) Score indexing lists must be sorted to guarantee the correctness of halting based on monotonic assumption of the TA or the CA based search strategy.

3.3 Two-Level Inverted Index

To handle the above problems, a two-level inverted index based on the sub-unit decomposition is constructed.

3.3.1 The Upper-Level Inverted Index

Given a database with graphs and their sub-unit representations, an inverted index can be constructed. For example, given a database of g_1 and g_2 in Figure 3.2, we can construct an inverted index for all sub-units derived from data graphs in this database as shown in Figure 3.5. This index is made up of two main parts: an index for all distinct sub-units from the given database, and an inverted list below each unit. Here, the sub-units are sorted in alphabetical order. Each entry in the inverted lists contains the graph identity and the frequency of the corresponding unit. All lists are sorted in increasing order of the graph size. In Figure 3.5, since $|g_1| < |g_2|$, g_1 is located before g_2 in the lists.

st_0: abbcc	
gid	freq
g_1	1

st_1: abbccd	
gid	freq
g_2	1

st_2: bab	
gid	freq
g_1	1
g_2	1

st_3: babcc	
gid	freq
g_1	1

st_4: babccd	
gid	freq
g_2	1

st_5: cab	
gid	freq
g_1	2
g_2	2

st_6: dab	
gid	freq
g_2	1

Figure 3.5: Upper-level inverted index for graphs

With this index, it is very convenient to fetch out graphs that contain a given sub-unit. Then, given a query, if we can quickly access sub-units that are highly similar to the sub-units from the query in increasing dissimilarity, graphs can also be accessed in globally increasing dissimilarity to the query. Therefore, a lower-level index for sub-units is built.

3.3.2 The Lower-Level Inverted Index

We construct the lower-level inverted index for all sub-units based on vertex labels. A sub-unit is broken into a multiset of labels excluding its root label. For example, st_0 in Figure 3.5 is decomposed into $\Psi_{st_0} = \{b, b, c, c\}$. With this decomposition, it is easy for us to build an inverted index for sub-units based

a	
sid	freq
st_2	1
st_5	1
st_6	1
st_3	1
st_4	1

b	
sid	freq
st_2	1
st_5	1
st_6	1
st_0	2
st_3	1
st_1	2
st_4	1

c	
sid	freq
st_0	2
st_3	2
st_4	2

d	
sid	freq
st_1	1
st_4	1

sid	size
st_2	2
st_5	2
st_6	2
st_0	4
st_3	4
st_1	5
st_4	5

Figure 3.6: Lower-level inverted index for sub-units

on labels. The index also contains two components: a label index in increasing order and inverted lists below labels recording the sub-unit identities and the frequencies of corresponding labels in the leaves of the sub-unit. Entries in each list are first grouped based on the leaf size of $|\Psi_{st}|$ and then sorted in decreasing frequencies within each group. For example, in Figure 3.6, the list below label b has three groups sorted in increasing leaf size. In the first group, st_2, st_5 and st_6 all have leaf sizes of 2. In the second group, st_0 and st_3 have leaf sizes of 4. In the last group, st_1 and st_4 have leaf sizes of 5. In each group, frequencies are sorted decreasingly. Considering in the last group, the frequency of st_1 is 2 which is larger than that of st_4 (=1). Moreover, the last list without a label index is an extended list storing the sizes of all sub-units in increasing leaf size.

With this index, it is convenient to search similar sub-units for a query sub-unit based on the sub-unit edit distance. We will present the details of the search algorithm in next section.

3.3.3 Index Maintenance

While employing a more complex two-level index in this work, it is worth noting that both these levels are inverted indexes and the features like sub-units and labels can be easily generated from individual graphs. As observed in [76], such inverted indexes can be implemented either with a special purpose inverted list engine or in commercial relational database systems. For the latter case, we will be building on various query optimization, concurrency control techniques that had been developed over the years [1] to update our indexes. For the earlier case, we will describe our operations here.

There are essentially seven kinds of updates for graph data: (1) inserting a new graph, (2) deleting a data graph, (3) inserting an edge into a graph, (4) deleting an edge of a graph, (5) inserting a new vertex into a graph, (6) deleting a vertex from a graph, and (7) relabelling a vertex in a graph. To support these updates, four kinds of operations occur in our two-level inverted index:

1. *Op*1: Inserting or deleting the graph information into an inverted list below a sub-unit in the upper-level index.

[1]This approach is also adopted in [24] where qgrams are stored in a relational database to support approximate string join.

2. *Op*2: Inserting or deleting the sub-unit information in an inverted list below a label in the lower-level index.

3. *Op*3: Create a new list for a new generated unit, or delete a unit from the upper-level index when its list is empty.

4. *Op*4: Create a new list for a new label, or delete a label from the lower-level index when its list is empty.

Assuming that the inverted index is properly implemented and optimized over a B-tree (or B$^+$-tree) [11], all the operations above will take at most $O(\log N)$ page accesses. Building on these operations, our index can easily support various types of updates as below: 1) Inserting a graph needs us to decompose this graph into a multiset of sub-units, and then perform *Op*1. For a new generated unit, we will perform *Op*3 followed by *Op*2. If a new label is detected, perform *Op*4. 2) Deleting a graph requires us to remove all the graph information in the upper-level index. 3) Inserting or deleting an edge of a graph affects two sub-units. Therefore, the graph information below two original sub-units is removed and they are inserted into two new lists. Furthermore, sub-unit information is also updated in the lower-level index. 4) Inserting or deleting a vertex only affects one unit. The operations are similar to update 3). 5) Relabelling a vertex will affect the sub-unit rooted by this vertex and those sub-units rooted by its neighbors. These operations are similar to updates 3) and 4).

3.4 Graph Similarity Search Algorithm

Based on the proposed two-level inverted index, we develop **SEGOS**, a cascade query processing framework, to employ the **dynamic mapping distance computation** and the **filtering strategy** proposed in Section 3.2 to enhance the graph search. The novel framework contains two search steps: the top-k sub-unit search and the graph similarity search. As shown in Figure 3.7, in the lower level, top-k similar sub-units to each sub-unit of the query can be returned quickly by using the TA search algorithm; in the upper level, graph pruning is done based on the top-k results from the lower level. To support continuous graph pruning, the CA graph search algorithm can be further divided into two stages: sorted list processing and dynamic graph mapping distance computation. In this step, sub-units for each data graph can be output with round-robin scan through the score sorted lists, and used as input to run dynamic mapping distance computation for seen data graphs with the query. This section will show how this framework work for graph pruning, and TA, CA, and DC denote the three stages in our framework.

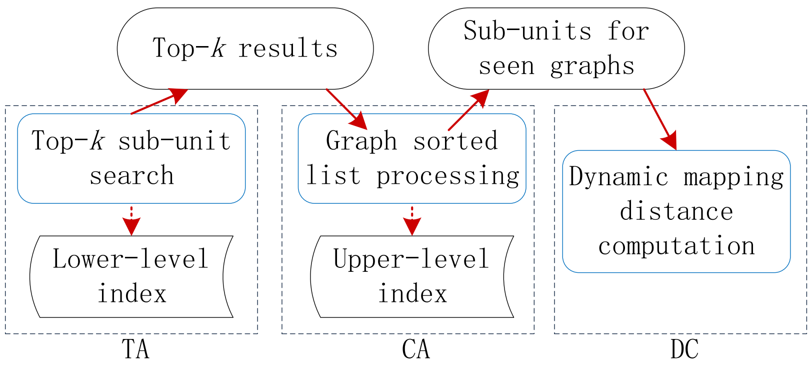

Top-k results
Sub-units for seen graphs
Top-k sub-unit search
Graph sorted list processing
Dynamic mapping distance computation
Lower-level index
Upper-level index
TA
CA
DC

Figure 3.7: The cascade search framework

3.4.1 Top-k Sub-unit Query Processing Algorithm

Given a query graph q, we need to efficiently find sub-units that are highly similar to each sub-unit from q in the TA stage. A full scan of the database to compute the sub-unit edit distance (STED) between each sub-unit and a query sub-unit can be very expensive. In this work, we propose a *top-k sub-unit searching algorithm* based on TA method [15]. The TA filtering strategy can help to avoid access to sub-units with high dissimilarity to the query sub-unit, but the score-sorted lists constructed need to guarantee the correctness of the TA halting monotonic assumption. From Definition 3.1 in Section 3.2.1, the STED between a query sub-unit st_q and any database sub-unit st_i can be represented as below.

$$\lambda(st_q, st_i) = \begin{cases} T(r_q, r_i) + 2 * |L_q| - (\psi + |L_i|), \ if \ |L_i| \leq |L_q| \\ T(r_q, r_i) - |L_q| - (\psi - 2 * |L_i|), \ if \ |L_i| > |L_q| \end{cases} \tag{3.1}$$

where $\psi = |\Psi_{L_q} \cap \Psi_{L_i}|$ denotes the common leave labels between st_q and st_i. From the above two equations, if we ignore the difference between the roots of sub-units $T(r_q, r_i)$, the STED increases when the value of $(\psi + |L_i|)$ or

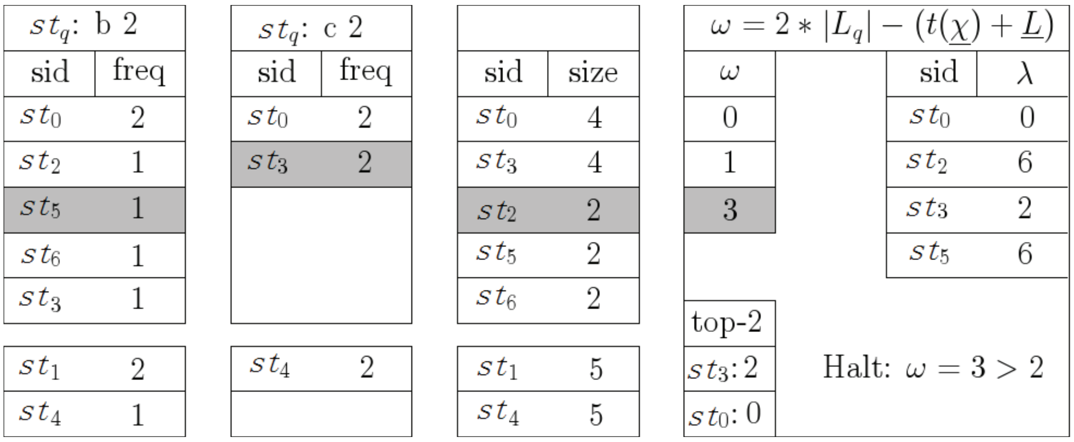
$(\psi - 2*|L_i|)$ decreases. Therefore, two aggregation functions can be derived as $\omega = 2*|L_q| - (\psi + |L_i|)$ and $\omega = -|L_q| - (\psi - 2*|L_i|)$ and we need to construct two sets of score-sorted lists to apply the above two functions. That means, sub-units with leaf sizes no more than $|L_q|$ and those with leaf sizes larger than $|L_q|$ must be processed separately.

Fortunately, the lower-level index can be used to conveniently construct these two sets of score-sorted lists. We know that each lower-level index list has been grouped increasingly according to sub-units' leaf sizes. Maintaining a leaf size array denoted by AL pointing to positions of all leaf size groups, it is easy to find the position that after which the leaf sizes are larger than that of the query sub-unit in $O(\log |AL|)$ time. Since each group has been sorted based on decreasing frequencies, all groups within a leaf size range can be directly merged into one list in $O(|AL| \times |SL|)$ time ($|SL|$ is the maximum length of all leaf size groups). Generally, $|AL|$ is a constant smaller number compared to $|SL|$, so the merge complexity can be considered to be linear. The detail of the merge function is given in Algorithm 3.1.

Algorithm 3.1: Merge function

```
Require: A list $L$ and a size index array $A$ of length $n$
Ensure: A score-sorted list $SL$
 1: $end \leftarrow true$, $max \leftarrow 0$, $p \leftarrow 0$
 2: initialize an array $A'$ with values of $A$;
 3: while $true$ do
 4:    for $i = 0$ to $n - 1$ do
 5:       if $A'[i] == A[i+1]$ then
 6:          continue;
 7:       $end \leftarrow false$;
 8:       if $max < L[A'[i]].freq$ then
 9:          $max \leftarrow L[A'[i]].freq$;
10:          $p \leftarrow i$;
11:    if $end == true$ then
12:       break;
13:    $SL.push\_back(L[A'[p]])$;
14:    $A'[p]++$;
```

Figure 3.8 shows the score-sorted lists obtained for $st_q = abbcc$ using the index in Figure 3.6. The query sub-unit st_q has leaf labels b and c. For the label b, we fetch out the inverted list under "b" in Figure 3.6. Then a size bound larger than $|L_q| = 4$ can be found in position 5 pointing to $(st_1, 2)$. From



st_q: b 2	
sid	freq
st_0	2
st_2	1
st_5	1
st_6	1
st_3	1
st_1	2
st_4	1

st_q: c 2	
sid	freq
st_0	2
st_3	2
st_4	2

sid	size
st_0	4
st_3	4
st_2	2
st_5	2
st_6	2
st_1	5
st_4	5

$\omega = 2 * |L_q| - (t(\underline{\chi}) + \underline{L})$

ω
0
1
3

sid	λ
st_0	0
st_2	6
st_3	2
st_5	6

top-2
st_3: 2
st_0: 0

Halt: $\omega = 3 > 2$

Figure 3.8: A top-k sub-unit searching example for $st_q = abbcc$

here, groups with leaf sizes no larger than 4 are merged into one single list of $\{(st_0, 2), (st_2, 1), (st_5, 1), (st_6, 1), (st_3, 1)\}$. Another list of $\{(st_1, 2), (st_4, 1)\}$ with leaf size larger than 4 is also formed. Similarly, two lists below c are formed as $\{(st_0, 2), (st_3, 2)\}$ and $\{(st_4, 2)\}$. The size list is also split into two parts, but the one with leaf sizes no larger than 4 should be reversely accessed decreasingly.

Given the score-sorted lists for st_q, suppose st_q has m distinct leaf labels with frequencies of $(c_1, c_2, \ldots, c_m)$. We compute $\psi = t(\chi)$ as the number of common leaf labels between st_q and any st_i.

$$t(\chi) = \sum_{j=1}^{m} \min\{c_j, \chi_j\}$$

where χ_j represents the frequency corresponding to st_i in the j^{th} score list of st_q. If st_i does not appear in that list, $\chi_j = 0$.

As shown in Alogirhtm 3.2, given m distinct label sorted lists and one size sorted list for st_q, the steps of our searching algorithm are:

1. Do sorted access in a round-robin schedule to each sorted list. If a sub-unit st_i is seen, compute $\lambda(st_q, st_i)$. Maintain a queue of top-k sub-units with the lowest λ values.

2. For each label list SL_j, let $\underline{\chi}_j$ be the frequency last seen under sorted access. Let $\underline{L}$ be the size last seen in the size list. For the score-sorted lists with smaller size, $\omega = 2 * |L_q| - (t(\underline{\chi}) + \underline{L})$. Otherwise, $\omega = -|L_q| -$

$(t(\underline{\chi}) - 2 * \underline{L})$. If the top-$k$ values are at most equal to ω, then halt. Otherwise, go to step 1.

Algorithm 3.2: Top-k sub-unit searching algorithm

```
Require: $m$ sorted lists $SL$ and 1 size list $L$ for $st_q$; $low$
Ensure: The top-$k$ similar sub-units
 1: $top - k \Leftarrow \emptyset$;
 2: for all sorted lists with $j = 1 \dots m+1$ do
 3:    if $j \le m$ then
 4:       $st_{id} \Leftarrow SL_j.getNext()$;
 5:       $\underline{\chi}_j \Leftarrow st_{id}.freq$;
 6:    else
 7:       $st_{id} \Leftarrow L.getNext()$;
 8:       $\underline{L} \Leftarrow st_{id}.size$;
 9:    if $st_{id}$ is not seen before then
10:       calculate $\lambda(st_q, st_{id})$;
11:       if $|top - k| < k$ then
12:          Maintain $top - k$ and continue;
13:       if $\lambda(st_q, st_{id}) < max\{\lambda | \lambda \in top - k\}$ then
14:          Maintain new $top - k$;
15:    if $low$ is true then
16:       $\omega = 2 * |L_q| - (t(\underline{\chi}) + \underline{L})$;
17:    else
18:       $\omega = -|L_q| - (t(\underline{\chi}) - 2 * \underline{L})$;
19:    if $\omega \ge max\{\lambda | \lambda \in top - k\}$ then
20:       return $top - k$;
21: return $top - k$;
```

Now we show the correctness of Algorithm 3.2 as below.

Proof. We show that the algorithm really returns the exact top-k result to a query sub-unit st_q when halting. Suppose we have m sorted lists for st_q. In fact, this algorithm can halt on two conditions:
1) The value of ω is no less than the maximum value in top-k. Since the top-k queue is maintained by the top-k minimum STEDs to st_q, when halting, they are naturally the top-k values among all sub-units having been retrieved. If we can prove that all remaining unseen sub-units have STEDs no less than the maximum value in top-k, the result is sure to be correct.

1.1) When processing the lists with smaller size graphs, we have

$$\omega = 2 * |L_q| - (t(\underline{\chi}) + \underline{L}) \geq max\{top - k\}$$

For any unseen sub-unit st_i, we have

$$\lambda(st_q, st_i) = T(r_q, r_i) + 2 * |L_q| - (t(\chi) + |L_i|)$$

where $T(r_q, r_i) \geq 0$. Since all lists in this case are sorted in decreasing orders, we have

$$t(\underline{\chi}) + \underline{L} = \sum_{j=1}^{m} \underline{\chi}_j + \underline{L} \geq t(\chi) + |L_i|$$

where all $\chi_x \in \chi$ and L_i are located below the halting positions. Therefore, $\omega \leq \lambda(st_q, st_i)$, i.e., unseen sub-units have $\lambda \geq \omega \geq max\{top{-}\mathrm{k}\}$. The top-$k$ results are the real k minimum values.

1.2) When running on sorted list with larger size graphs, we have

$$\omega = -|L_q| - (t(\underline{\chi}) - 2 * \underline{L}) \geq max\{top - k\}$$

In this case, for any unseen sub-unit st_i, we have

$$\lambda(st_q, st_i) = T(r_q, r_i) - |L_q| - (t(\chi) - 2 * |L_i|)$$

where $T(r_q, r_i) \geq 0$. Since label lists are sorted decreasingly while size list is sorted increasingly, we have

$$t(\underline{\chi}) - 2 * \underline{L} = \sum_{j=1}^{m} \underline{\chi}_j - 2 * \underline{L} \geq t(\chi) - 2 * |L_i|$$

where all $\chi_x \in \chi$ and L_i are located below the halting positions. Therefore, $\omega \leq \lambda(st_q, st_i)$, i.e., unseen sub-units have $\lambda \geq \omega \geq max\{top{-}\mathrm{k}\}$. The top-$k$ results are correct to be the k minimum values.

2) Algorithm halts when all sorted lists have been accessed to the ends. In this case, with post processing, the top-k result is sure to be correct because they are the k minimum values among all sub-units. □

Previous Figure 3.8 also shows an example to search top-2 similar sub-units to $st_q = abbcc$ on score-sorted lists containing sub-units with lower leaf sizes.

Sub-units are accessed in a round-robin way from the list below label b to the size list. STED is calculated for each sub-unit seen and a top-2 queue is maintained. Algorithm halts in the positions with gray shadows because $\omega = 2 * 4 - (1 + 2 + 2) = 3 \geq 2$, where 2 is the maximum value in the top-2 queue. Obviously, the top-2 results are returned without access to st_6.

3.4.2 Score-Sorted Lists Construction

The above algorithm provides us an efficient way to return highly similar sub-units to a query sub-unit. Then graph score sorted lists can be easily formed by combining a set of lists fetched from the upper-level index below the corresponding top-k results.

Given a query graph q, for each query sub-unit st_q, its top-k queue is returned from the lower-level TA stage. Then, for each sub-unit st_i in the queue, a graph inverted list indexed by st_i can be directly fetched from the upper-level index. Therefore, k graph lists will be returned for each query sub-unit st_q. Later the k graph lists will be split into two segments: those with graph sizes larger than $|q|$, and those not. Segments within a graph size range will be combined into one group. Within each group, graphs are naturally ordered in terms of STEDs according to the top-k values. Furthermore, in the group with smaller sizes, the segments having STED larger than $\lambda(st_q, \epsilon)$ are discarded. Since the upper-level index lists have been sorted by increasing graph sizes, finding size range position takes $O(\log |GL|)$ time ($|GL|$ is the maximum size of all graph size index arrays).

For example, given a query $q = g_1$ in Figure 3.1, the top-2 similar sub-units for the query sub-unit st_5 are st_5 and st_2, in Figure 3.9. Then two graph lists indexed by st_5 and st_2 are extracted from the upper-level index in Figure 3.5: $\{(g_1, 2), (g_2, 2)\}$ and $\{(g_1, 1), (g_2, 1)\}$. Since the query is of size 5, each graph list is divided into two segments. For example, the list below st_5 is split into $\{(g_1, 2)\}$ with $|g_1| \leq 5$ and $\{(g_2, 2)\}$ with $|g_2| > 5$. Similarly, the list below st_2 is split into $\{(g_1, 1)\}$ and $(g_2, 1)\}$. After that, segments $\{(g_1, 2)\}$ and $\{(g_1, 1)\}$ with smaller sizes are combined into one list $\{(g_1, 2), (g_1, 1)\}$. Since $\lambda(st_5, st_5) = 0 \leq \lambda(st_5, st_2) = 1$, $(g_1, 2)$ is located before $(g_1, 1)$. In Figure 3.9, if a graph is fetched from a list below a sub-unit, it is connected to that sub-unit using a dashed arrow.

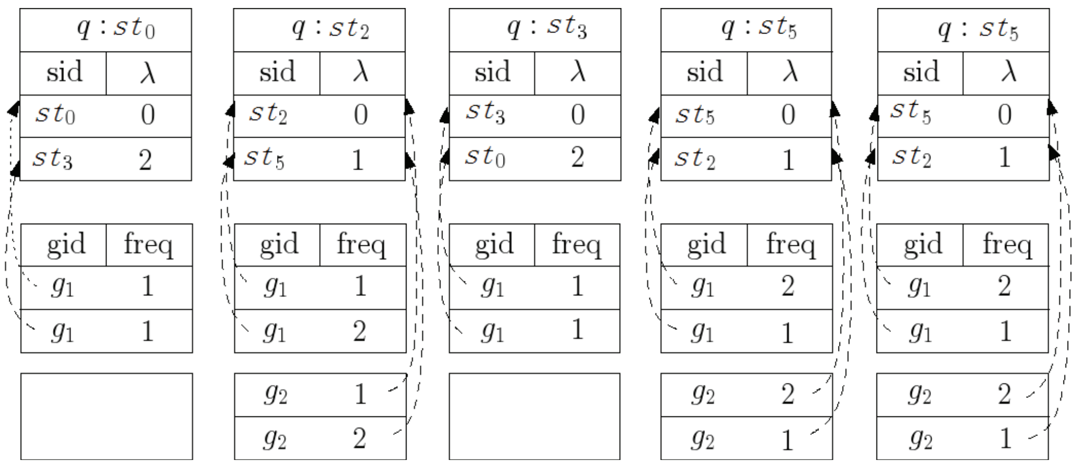


Figure 3.9: The sorted lists for $q = g_1$

Based on the constructed graph score sorted lists, the CA stage accesses sub-units for data graphs using a round-robin scan. Using the summation of STEDS as an aggregation function, the halting condition and several aggregation bounds can be directly derived.

3.4.3 Bounds from Aggregation Function

Given m score lists of a query graph q, we compute the overall score of a graph g having been seen, denoted by $\zeta(q, g)$ as

$$\zeta(q, g) = t'(\chi_1, \ldots, \chi_m) = \sum_{j=1}^{m} \chi_j$$

χ_j is a local minimum STED of graph g having been seen below the j^{th} list of q. The computation of χ_j is as below.

Definition 3.2. *Let $Se_j = \{e_1, \ldots, e_x\}$ including all STEDs of a graph g below the j^{th} list. Then the corresponding χ_j of g is computed as*

$$\chi_j = \min_{e_i \in Se_j} \{e_i\}$$

Generally, if Se_j is empty, $\chi_j = 0$.

Example 3.1. *As shown in Figure 3.10, a graph g_1 has been seen blow three lists GL_1, GL_2, and GL_4 of q (this can be seen in cells with slashes in the* *figure). We have its local minimum STED in each list as $\chi_1 = 0$, $\chi_2 = 0$, and $\chi_4 = 1$. Since Se_3 is empty, $\chi_3 = 0$. Therefore, the overall score of g_1 obtained from q is $\zeta(q, g_1) = 0 + 0 + 0 + 1 = 1$.*

GL_1		
$q : st_0$		
gid	sid	λ
g_1	st_0	0
g_2	st_3	3
g_3	st_1	4
g_4	st_1	4
...	...	...

GL_2		
$q : st_0$		
gid	sid	λ
g_1	st_0	0
g_2	st_3	3
g_3	st_1	4
g_4	st_1	4
...	...	...

GL_3		
$q : st_2$		
gid	sid	λ
g_2	st_2	0
g_2	st_5	2
g_3	st_5	2
g_5	st_6	2
...	...	...

GL_4		
$q\ st_3$		
gid	sid	λ
g_3	st_3	0
g_1	st_7	1
g_2	st_5	3
g_1	st_4	4
...	...	...

Figure 3.10: An example for computing CA bounds

Suppose $l(g) = \{l_1, \dots, l_y\} \subseteq \{1, 2, \dots, m\}$ is a set of known lists of g having been seen below q. Let $\underline{\chi}(g)$ be the multiset of distances corresponding to the distinct sub-units of g last seen.

- **Aggregation Lower Bound** denoted by $L_\mu(q, g)$ is obtained by substituting the missing lists $j \in \{1, 2, \dots, m\} \setminus l(g)$ with $\underline{\chi}_j$ (the distance last seen under the j^{th} list) in $\zeta(q, g)$. That is, $\chi_j = \underline{\chi}_j$ when Se_j is empty.

- **Aggregation Upper Bound** denoted by $U_\mu(q, g)$ is computed as $U_\mu(q, g) = t'(\underline{\chi}(g)) + \overline{\chi} * (max\{|q|, |g|\} - |\underline{\chi}(g)|)$.

Here, $\overline{\chi} = max_{st \in S(q) \cup S(g)}\{\lambda(st, \epsilon)\}$. As shown in Example 3.1, we have $\zeta(q, g_1) = 1$. Suppose the cells with gray shadows are the current positions accessed, the distances last seen below the lists of q is $\{4, 3, 2, 1\}$. To replace the unseen value χ_3 of g_1 with $\underline{\chi}_3 = 2$, $L_\mu(q, g_1) = \zeta(q, g_1) + \underline{\chi}_3 = 1 + 2 = 3$. It can be seen from Figure 3.10, the distinct sub-unit set of g_1 last seen is $\underline{\chi}(g_1) = \{st_0, st_7\}$. Suppose $|g_1| = 3$, a remaining sub-unit st_4 has not been accessed (the cell with back slash), and the maximum distance between sub-units in q and g_1 is $\overline{\chi} = max_{st \in S(q) \cup S(g_1)}\{\lambda(st, \epsilon)\} = 11$. To substitute the value of unseen sub-units from g_1 to q with $\overline{\chi}$, $U_\mu(q, g_1) = t'(\underline{\chi}(g_1)) + \overline{\chi} * (max\{|q|, |g_1|\} - |\underline{\chi}(g_1)|) = 0 + 1 + 11 * (4 - 2) = 23$.

Theorem 3.2. *Let g_1 and g_2 be two graphs, the bounds obtained as above satisfy the following:*

$$\zeta(g_1, g_2) \leq L_\mu(g_1, g_2) \leq \mu(g_1, g_2) \leq U_\mu(g_1, g_2)$$

Proof. From the aggregation bounds definitions in Section 3.4.3, it is clear that $\zeta(g_1, g_2) \leq L_\mu(g_1, g_2) \leq U_\mu(g_1, g_2)$. Now we prove $L_\mu(g_1, g_2) \leq \mu(g_1, g_2)$. Suppose P is an optimal alignment between $S(g_1)$ and $S(g_2)$. Then,

$$\mu(g_1, g_2) = \sum_{st_i \in S(g_1)} \lambda(st_i, P(st_i))$$

where $P(st_i)$ is each sub-unit in g_2 aligned to st_i in g_1 and $P(st_i) \in S(g_2) \cup \{\varepsilon\}$. Let $\zeta(g_1, g_2)$ of g_2 be the overall score obtained by computing the summation of all local minimum STED of g_2 below m sorted lists for g_1.

1) For those lists below $S'(g_1)$ including entries of g_2, since they contain the top-k lowest scores, we have

$$\begin{aligned}\sum_{st_i \in S'(g_1)} \min_{e_i \in Se}\{e_i\} &= \sum_{st_i \in S'(g_1)} \min_{st_j \in S(g_2)}\{\lambda(st_i, st_j)\} \\ &\leq \sum_{st_i \in S'(g_1)} \lambda(st_i, P(st_i))\end{aligned}$$

2) For those below $S''(g_1) = S(g_1) \backslash S'(g_1)$:

$$\begin{aligned}\sum_{st_i \in S''(g_1)} \min\{\underline{\chi}_i, \lambda(st_i, \varepsilon)\} &\leq \sum_{st_i \in S''(g_1)} \min_{st_j \in S(g_2) \cup \{\varepsilon\}}\{\lambda(st_i, st_j)\} \\ &\leq \sum_{st_i \in S''(g_1)} \lambda(st_i, P(st_i))\end{aligned}$$

Accordingly, we obtain $L_\mu(g_1, g_2)$ and $\mu(g_1, g_2)$ as,

$$L_\mu(g_1, g_2) = \sum_{st_i \in S'(g_1)} \min_{e_i \in Se}\{e_i\} + \sum_{st_i \in S''(g_1)} \min\{\underline{\chi}_i, \lambda(st_i, \varepsilon)\}$$

$$\mu(g_1, g_2) = \sum_{st_i \in S'(g_1)} \lambda(st_i, P(st_i)) + \sum_{st_i \in S''(g_1)} \lambda(st_i, P(st_i))$$

Therefore, $L_\mu(g_1, g_2) \leq \mu(g_1, g_2)$.

3) We prove $U_\mu(g_1, g_2) \geq \mu(g_1, g_2)$. As described in **Aggregation Upper**

Bound, $\underline{\chi}(g_2)$ is a multiset of distances corresponding to the sub-units of g_2 last seen in known lists without duplicates, and

$$\overline{\chi} = max_{st \in S(g_1) \cup S(g_2)} \{\lambda(st, \epsilon)\}$$

Suppose $S'(g_2) \subseteq S(g_2)$ is the sub-units corresponding to $\underline{\chi}(g_2)$, and $S'(g_1)$ contains sub-units of g_1 aligned to $S'(g_2)$ due to $\underline{\chi}(g_2)$. If $S'(g_2) \subseteq \{P(st_i) | st_i \in S'(g_1)\}$, we have

$$t'(\underline{\chi}(g_2)) = \sum_{st_i \in S'(g_1)} \lambda(st_i, P(st_i))$$

$$\overline{\chi} * (max\{|g_1|, |g_2|\} - |\underline{\chi}(g_2)|) \geq \sum_{st_i \in S(g_1) \setminus S'(g_1)} \lambda(st_i, P(st_i))$$

If $S'(g_2) \nsubseteq \{P(st_i) | st_i \in S'(g_1)\}$, we have

$$t'(\underline{\chi}(g_2)) \geq \sum_{st_i \in S'(g_1)} \lambda(st_i, P(st_i))$$

$$\overline{\chi} * (max\{|g_1|, |g_2|\} - |\underline{\chi}(g_2)|) \geq \sum_{st_i \in S(g_1) \setminus S'(g_1)} \lambda(st_i, P(st_i))$$

Accordingly, we obtain $U_\mu(g_1, g_2) \geq \mu(g_1, g_2)$. □

3.4.4 Graph Pruning Algorithm

Our graph pruning algorithm is a CA-based algorithm. Its filtering strategy is similar to the top-k sub-unit search, while using a different aggregation function. It also employs the above aggregation bounds and dynamic mapping distance computation algorithm to reduce the graph mapping distance computation. A simple example of graph sorted lists processing can be seen in Figure 3.4 in Section 3.2. The detail of our CA-based algorithm are shown in Algorithm 3.3.

Given m sorted lists for a graph query q and a threshold τ, the main steps are shown as below:

1. Perform sorted retrieval in a round-robin schedule to each sorted list. At each depth h of lists:

   - Maintain the lowest values $\underline{\chi}_1, \ldots, \underline{\chi}_m$ encountered in the lists. Maintain a distance accumulator $\zeta(q, g_i)$ and a multiset of retrieved sub-units $S'(g_i) \subseteq S(g_i)$ for each g_i seen under lists.

- For each g_i that is retrieved but unprocessed, if $\zeta(q, g_i) > \tau * \delta_{g_i}$ ($\delta_{g_i} = \max\{4, [\max\{\delta(q), \delta(g_i)\} + 1]\}$), filter out the graph; if $L_\mu(q, g_i) > \tau * \delta_{g_i}$, filter out the graph; if $U_\mu(q, g_i) \leq \tau * \delta_{g_i}$, add the graph to the candidate set. Otherwise, if $\mu(S(q), S'(g_i)) > \tau * \delta_{g_i}$, filter out the graph. If all the above bounds are useless, run the Dynamic Hungarian algorithm to obtain $L_m(q, g_i)$ and $U_m(q, g_i)$ for filtering.

2. When a new distance is updated, compute a new ω. If $\omega = t'(\underline{\chi}) = \sum_{j=1}^{m} \underline{\chi}_j > \tau * \delta'$, then halt. Otherwise, go to step 1.

Algorithm 3.3: CA-based range query algorithm

```
Require: $m$ sorted lists $GL$ for $q$, $\tau$ and $h$
Ensure: All $g_i$ s.t. $\lambda(q, g_i) \leq \tau$
 1: $candidate \Leftarrow \emptyset$; $flag \Leftarrow false$;
 2: for all sorted lists $GL_j$ with $j = 1 \ldots m$ do
 3:    $g_{id} \Leftarrow GL_j.getNext()$;
 4:    $\underline{\chi}_j \Leftarrow g_{id}.dist$;
 5:    maintain the distance accumulator $\zeta(q, g_{id})$;
 6:    maintain the multiset for seen sub-units $S'(g_{id})$;
 7:    if $scandepth \% h == 0$ then
 8:       for all $g_{id}$ seen and unprocessed do
 9:          if $\zeta(q, g_{id}) > \tau * \delta_{g_i}$ then
10:             filter it out and continue;
11:          if $L_\mu(q, g_{id}) > \tau * \delta_{g_i}$ then
12:             filter it out continue;
13:          if $U_\mu(q, g_{id}) > \tau * \delta_{g_i}$ then
14:             further compute other bounds;
15:          if $\mu(S(q), S'(g_{id})) > \tau * \delta_{g_i}$ then
16:             filter it out and continue;
17:          Filtering with $L_m(q, g_{id})$ and $U_m(q, g_{id})$;
18:    if $\omega = t'(\underline{\chi}) > \tau * \delta'$ then
19:       $flag \Leftarrow true$ and break;
20: if $flag \neq true$ then
21:    post process the remaining graphs not appeared;
```

The correctness of Algorithm 3.3 is shown as below.

Proof. We prove that our candidate set includes all positive results when algorithm halts.

1) The algorithm halts with $\omega > \tau * \delta'$.

1.1) Running on the sorted lists with smaller size graphs, entries in each list below $st_j \in q$ have distances $\chi_j \leq \lambda(st_j, \epsilon)$. From the halting condition, we have $\omega = t'(\underline{\chi}) > \tau * \delta'$. Then, for any unseen graph $g_i \subseteq D'$, suppose P is the optimal alignment between $S(q)$ and $S(g_i)$. From Definition 3.1, we have

$$\mu(q, g_i) = \sum_{st_j \in S(q)} \lambda(st_j, P(st_j))$$

where $P(st_j)$ is each sub-unit in g_i aligned to st_j in q and $P(st_j) \in S(g_i) \cup \{\varepsilon\}$. Since $|g_i| \leq |q|$, and g_i locates below halting positions of ω. We have $\underline{\chi}_j \leq \lambda(st_j, P(st_j))$. Hence, $\omega \leq \mu(q, g_i)$. Therefore, for any $g_i \subseteq D'$, we have

$$L_m(q, g_i) = \frac{\mu(q, g_i)}{\delta_{g_i}} \geq \frac{\mu(q, g_i)}{\delta'} \geq \frac{\omega}{\delta'} > \tau$$

Any unseen $g_j \subseteq D'$ can be safely filtered out.

1.2) Similarly, if algorithm runs on the sorted list with larger size graphs, any unseen $g_i \subseteq D'$ also can be safely filtered out.

2) Algorithm halts when the ends of all sorted lists have been reached. In this case, this algorithm will become a linear scan algorithm by postprocessing the remaining unseen graphs, which guarantees that we have the correct candidate set without false negative. □

Obviously, the CA method performs the pruning test only for every h index entries accessed, and aggregation bounds can be accumulated in constant time. For data graphs having very similar sub-units to the query, aggregation upper bounds are small enough to output them as candidates; while for those having very dissimilar sub-units, aggregation lower bounds are large enough to prune them. Therefore, aggregation bounds take negligible constant time for early filtering.

As described before, our whole search strategy includes the TA, CA, and DC stages. Previously, we have provided the complexity analysis of some steps. Here, we present a more complete analysis. First, in the TA stage, constructing sorted lists for each queried sub-unit is decided by the merge time, which takes $O(|AL| \times |SL|)$ time as shown in Section 3.4.1, and a simple study of the TA search complexity is in [15]. The worst case of this step takes $O(kd|SL|)$ (k is the value of top-k results and d is the average degree of sub-units) time for sorted access and takes $O(N \log k)$ (N is the number of graphs accessed) time for maintaining a heap. Second, in the CA stage, graph sorted lists are combined by top-k results. As stated in Section 3.4.2, it takes $O(\log |GL|)$ time. Third, in the DC stage, the CA search complexity is similar to the TA search. We compute the dynamic mapping distance in $\Theta(n^3)$ (n is the average size of graphs) time and do the sub-unit difference operation in $O(\log n)$ time.

3.4.5 Pipe-line Graph Similarity Search Algorithm

As shown in Figure 3.7, the above graph pruning algorithm can be divided into two stages: graph sorted list processing (CA) and dynamic graph mapping distance computation (DC). In step 1, we only use aggregation bounds, and output accessed graphs with seen sub-unit multisets to the separate DC stage for mapping distance computations. The main advantage of our query processing framework lies in reducing the complex GED bounds computations by avoiding accessing highly dissimilar graphs.

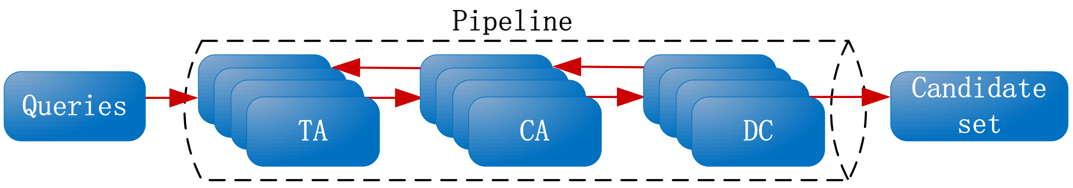


Figure 3.11: The pipeline of query processing framework

Moreover, the proposed approaches can be further improved by pipelining. It is easy to pipeline the whole query processing framework in Figure 3.7 into three consecutive stages: TA, CA, and DC. As shown in Figure 3.11, given graph queries, they are first decomposed into multiple sub-unit multisets. Then, each sub-unit is input to the TA stage to get its top-k similar sub-units. The output of top-k results for each query graph is fed to the input of the CA stage for building the graph score sorted lists. After that, the CA stage retrieves graph score sorted lists for each query graph in a round-robin schedule. When CA halts or the ends of all lists have been reached, all the accessed sub-units for seen data graphs are arranged to be the input of the DC stage. In the DC stage, we compute partial mapping distance when the accessed sub-units for the data graph are more than 50%, and run dynamic computation for graphs which have been processed but not filtered out. Moreover, there is no need to further return top-k results in the TA stage when the CA halts.

The pipelining algorithm can avoid parameter tunings for the k value in the TA stage and the h value in the CA stage. The k value can be fixed as a small number like 20, and h is not needed since the CA stage does not control the dynamic computations. To reduce dynamic computation overhead, we run partial matching only when more than 50% sub-units of a graph have been accessed. Further consideration will be illustrated in Section 4.6. Hereafter, **SEGOS** means our original CA search algorithm without pipeline, and **SEGOS-Pipeline** refers to the pipelining one.

3.5 Experimental Study

In this section, we compare our methods with two state-of-the-art approaches **C-Tree** [29] and **κ-AT** [63] on two real datasets. **SEGOS** was compiled with gcc 4.4.3 in Red hat Linux Operating System, and all experiments were run on a server with Quad-Core AMD Opteron(tm) Processor 8356, 128GB memory, running RHEL 4.7AS. In the experiments, we randomly selected 20 graphs from the dataset as query graphs and present the average result.

AIDS Dataset. This dataset is a DTP AIDS Antiviral Screen chemical compound dataset, published by National Cancer Institute[2]. This dataset has been widely used in many existing works [10, 22, 29, 63, 70, 78]. It consists of $42,687$ chemical compounds, with an average of 46 vertices. Compounds are labelled with 63 unique vertex labels.

Linux Dataset. *Program Dependence Graph (PDG)* is an ideal static representation of the data flow and control dependency within a procedure, with each vertex assigned to one statement and each edge representing the dependency between two statements. PDG is widely used in software engineering for clone detection, optimization, debugging, etc (e.g., [18, 19]). Here, we use CodeSurfer 2.1pl to generate the PDG dataset[3]. First we maximize the configuration of the Linux kernel and then dump the *Program Dependence Graph* using CodeSurfer 2.1pl with strict error limitation. This Linux kernel procedure dataset has in total 48,747 graphs, with an average of 45 vertices. The vertices of graphs are labelled with 36 unique labels, representing the roles of vertices

[2]http://dtp.nci.nih.gov/docs/aids/aids_data.html
[3]http://www.grammatech.com

in the procedure, such as "declaration", "expression", etc.

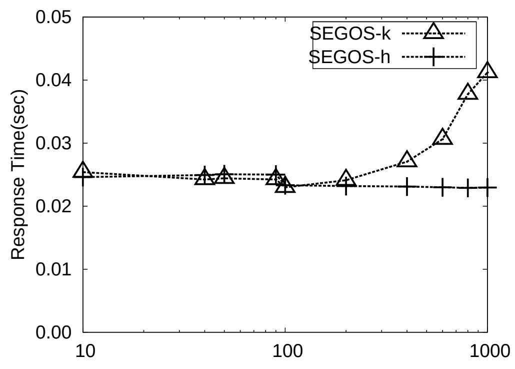
Taken from different applications, AIDS is a sparse database with near normal size distribution while Linux is that with near uniform size distribution. Table 3.1 presents five major parameters used in our experiments, including their descriptions and values (with default values in bold). Hereafter, the default values will be used in all the experiments if not particularly indicated.

Table 3.1: Parameter settings on graph similarity search

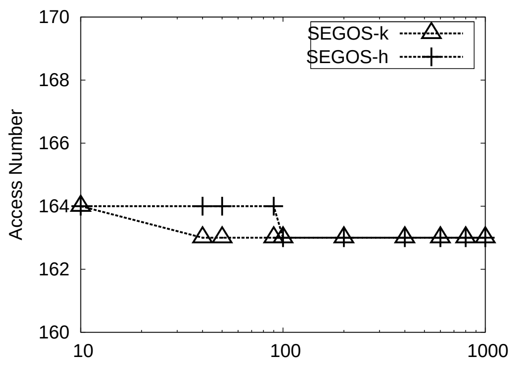
Parameter	**Description**	**Value**
k_s	k value for the TA stage	$10, 20, .., \mathbf{100}, 200, .., 1000$
h	h value for the CA stage	$10, 20, .., 100, 200, .., \mathbf{1000}$
$\|D\|$	dataset graph number	5K,10K,15K,**20K**,25K,..,40K
$\|q\|$	query vertex number	$10, 20, 30, 40, 50, 60, 70, 80$
τ	distance threshold	$0, 2, 4, 6, 8, \mathbf{10}, 12, 14, 16, 18, 20$

3.5.1 Sensitivity Study

We first conduct a series of parameter sensitivity analysis on our non-pipeline algorithm **SEGOS**. The impact of different parameters on the access number and the response time is presented. Access number here is defined as the number of graphs accessed to compute mapping distances for a query graph.



(a) Response time



(b) Access number

Figure 3.12: Sensitivity test on AIDS dataset

In Figure 3.12 **SEGOS-k** and **SEGOS-h** respectively correspond to the sensitivity of parameters k_s and h. It can be seen that, when k_s is small, the lists of top-k sub-units are quite short. In this case, our algorithm filters out few graphs after scanning through the lists, and the dynamic algorithm has to be applied on more sub-units for the remaining graphs. As k_s increases from 50 to 100, the lists of top-k sub-units become larger for the CA stage and more graphs are pruned off early. When k_s is larger than 100, there is little change in the access number, since CA has reached its halting conditions.

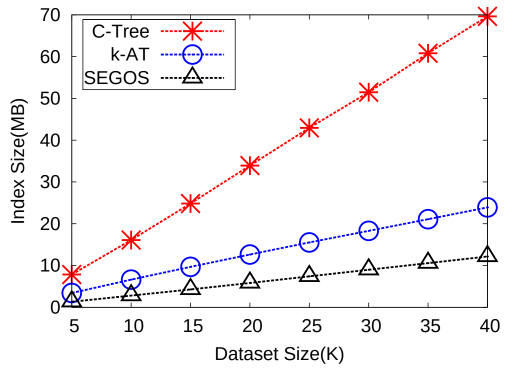
Meanwhile, when h is small, the chance of retrieving the whole set of sub-units is low. As h grows, more sub-units will be seen, allowing more graphs to be pruned without being fully accessed. As such, both the response time and the access number decrease as h increases from 10 to 100. The response time will be stable when h is large enough to hit the halting condition of CA stage.

We exclude results on the Linux dataset since it shows very similar trends. However, the sensitive values for k_s are larger in this dataset because its size distribution is more uniform than the AIDS dataset. Generally, our method achieves good performance by setting k_s as about 1% of the total sub-unit number and h as in the order of a few hundred. Without loss of generality, we simply use the default values in Table 3.1 for both two real datasets in the following experiments.

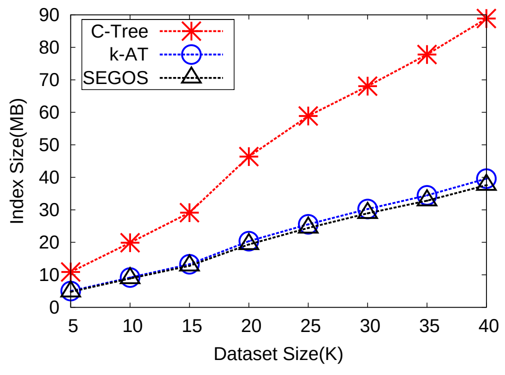
3.5.2 Index Construction Performance

In this subsection, we evaluate index construction performance of **SEGOS**, **κ-AT** and **C-Tree** w.r.t the dataset size. To build the index for **κ-AT**, we first conduct a sensitivity test and find that **κ-AT** performs the best by setting $\kappa = 2$.

Figure 3.13 and 3.14 show the index size and index construction time on both datasets, with $|D|$ varying from $5K$ to $40K$. We can see that **SEGOS** needs the shortest construction time and takes up the smallest space among all the three index structures, for it is sufficient for **SEGOS** to build two simple inverted indexes with only one dataset scan. For the other two index strategies, we find that **κ-AT** has to scan the dataset up to κ times to build a κ-layer feature table for each graph, and index these elements in all feature tables, and **C-Tree** uses one complex R-Tree like index structure, making it the most expensive one in index construction and the largest one in index size. In summary, **SEGOS** outperforms **κ-AT** and **C-Tree** in terms of index size and build time.

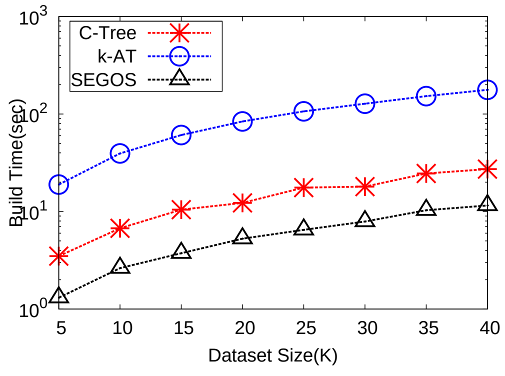


(a) On AIDS dataset

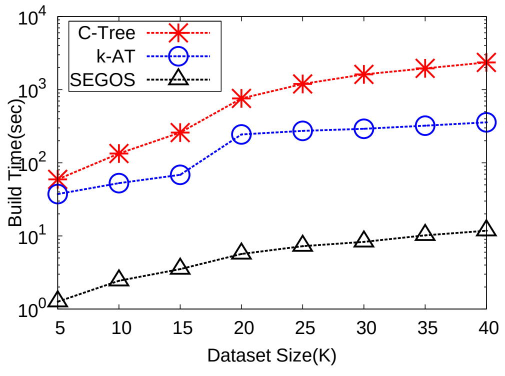


(b) On Linux dataset

Figure 3.13: Index size vs. $|D|$



(a) On AIDS dataset



(b) On Linux dataset

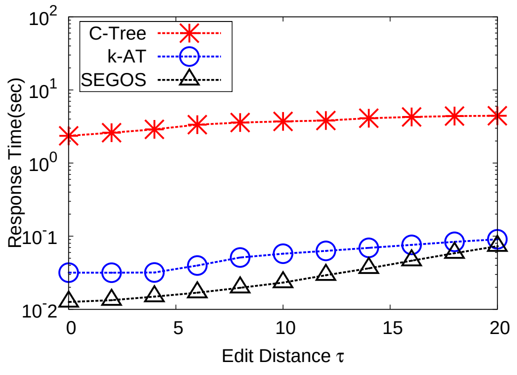
Figure 3.14: Construction time vs. $|D|$

3.5.3 Query Performance

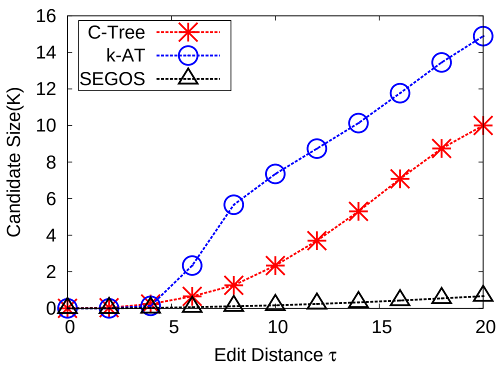
We next investigate the performance of our range query algorithms compared against those of **C-Tree** and **κ-AT**. Figure 3.15 and 3.16 show the results of range queries with τ varying from 0 to 20, $|D| = 20K$.

From Figure 3.15 we can see that **SEGOS** always returns the smallest number of candidates while incurring shortest response time. On the AIDS dataset, it outperforms **κ-AT** by up to two orders of magnitude in terms of candidate set size, and beats **C-Tree** in terms of filtering efficiency by two orders of magnitude.

Figure 3.16(b) shows that **κ-AT** has the poorest filtering ability, although it is the fastest one when it comes to filter as shown in Figure 3.16(a). Even when τ is as small as 6, **κ-AT** gives 800 more candidates than **SEGOS**. Here

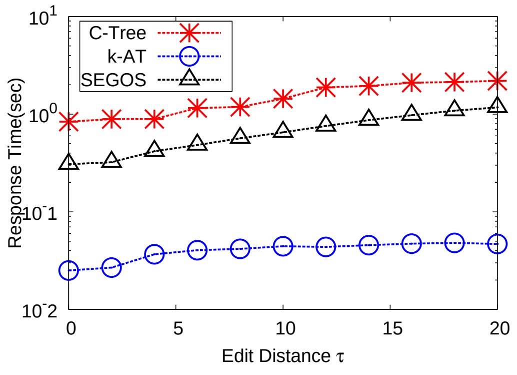


(a) Response time vs. τ

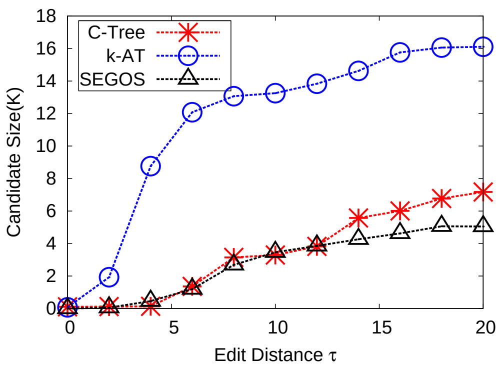


(b) Candidate size vs. τ

Figure 3.15: Range queries on AIDS dataset



(a) Response time vs. τ



(b) Candidate size vs. τ

Figure 3.16: Range queries on Linux dataset

we can conclude that although the simplistic filtering adopted by **κ-AT** gives it higher efficiency, it's filtering power is however much weaker than the other two. In a more concrete term, **κ-AT** is fast simply because it does not do much filtering. Compared to **C-Tree**, it is clear that **SEGOS** dominates **C-Tree** w.r.t response time and candidate size. The superiority of **SEGOS** becomes more significant when τ grows larger. We can see that **C-Tree** returns $2K$ more candidates than **SEGOS**, which is about 1/10 of the entire dataset size.

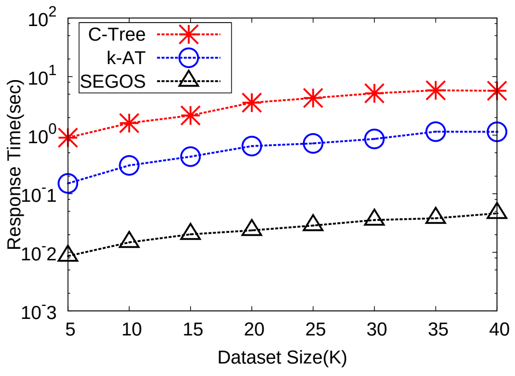
There are two reasons for the best result of our algorithm in Figure 3.15(a). First, the number of accessed graphs for mapping distance computation is much smaller on the AIDS dataset than the Linux dataset. Second, the randomly selected queries include graphs with smaller sizes or with high dissimilarity to most graphs in the AIDS dataset which can be fast completed in **SEGOS**.

Note that candidates verification using the GED is an extremely expensive process (NP-Hard). If we take into consideration that the GED computation for each of acquired candidates (of average size 40) is in thousand of seconds, then the extra candidates generated by κ-**AT** (eg. 800) will cost an additional hundreds of thousands seconds. From our observation, the total response time including filtering and verification time increases as the candidate number becomes larger. As such, it makes sense to sacrifice a little more time to filter out as many candidates as possible, as **SEGOS** does.

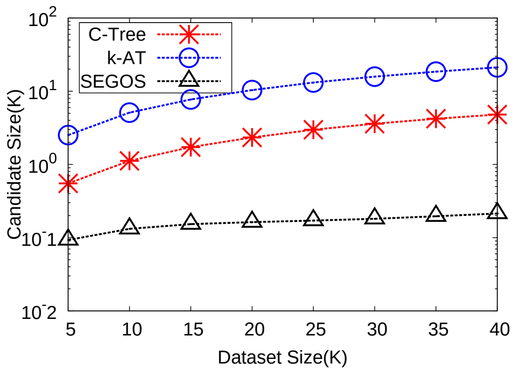
3.5.4 Scalability Study

We conduct two groups of experiments to evaluate the scalability of our algorithm in terms of the dataset size over two real datasets.

Figure 3.17 and 3.18 illustrate the scalability of the algorithms with respect to the dataset size $|D|$, ranging from $5K$ to $40K$. Here, we choose $\tau = 2$ for Linux dataset, and $\tau = 10$ for the AIDS dataset. This is because there are many similar graphs in the Linux dataset and a small τ is sufficient to show the difference in performance (**SEGOS** also performs better than the others when τ is large). On the contrary, since the AIDS dataset does not have that many similar graphs, a larger τ is more appropriate to reveal the difference.



(a) Response time vs. $|D|$

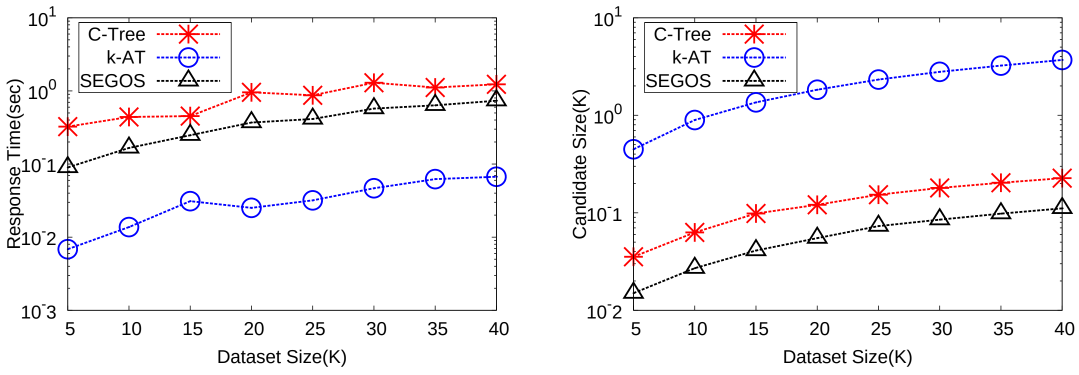


(b) Candidate size vs. $|D|$

Figure 3.17: Scalability of range queries on AIDS dataset

Figure 3.17 shows that **SEGOS** outperforms the other two algorithms over the entire range of dataset sizes. Furthermore, as the dataset size grows, **SEGOS**'s response time increases only from $8ms$ to $40ms$, which is only 0.1% that of **C-Tree** and 50% that of κ-**AT**. On the Linux dataset, **SEGOS** is still the most effective one in candidate filtering and costs moderate response time.

From these two figures, we can see that **SEGOS** is better than κ-**AT** on the AIDS dataset, and **C-Tree** on both the AIDS and the Linux datasets. Though **SEGOS** needs more time than κ-**AT** on the Linux dataset, it prunes more graphs than κ-**AT**, by two orders of magnitude.



(a) Response time vs. $|D|$ (b) Candidate size vs. $|D|$

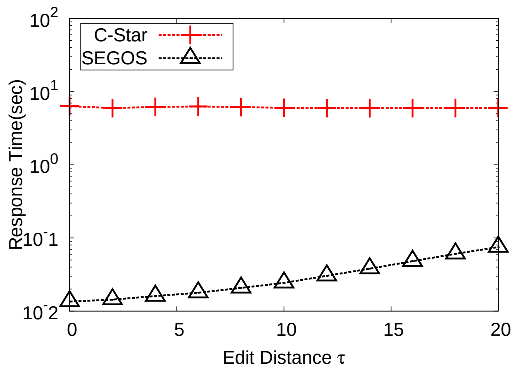
Figure 3.18: Scalability of range queries on Linux dataset

3.5.5 Effects of SEGOS on C-Star

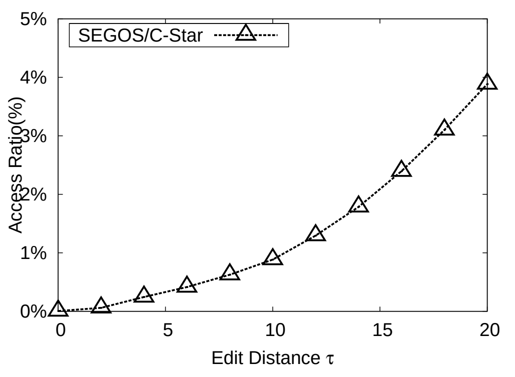
To show how much **SEGOS** can enhance **C-Star**, we conduct a set of experiments to see the response time and the access ratio of **SEGOS**, compared to **C-Star**. $20K$ graphs are randomly selected from two real datasets, and 10 graphs are extracted as queries. Figure 3.19 shows that **SEGOS** can enhance **C-Star** by dramatically reducing mapping distance computations by two orders of magnitude on average. We also investigate queries which have a mass of similar graphs in the database, since in this special case our method may degrade to the linear case of **C-Star** while taking extra overhead for the TA stage. However, we find that the overhead can be negligible, even in the worst case, this overhead takes less than 0.1% of the overall response time. A result showing the overhead with various k_s values is presented in Figure 3.20.

3.5.6 Effects of the Pipelining Algorithm

We also implement a simple pipelining algorithm for **SEGOS**, denoted by **SEGOS-Pipeline**. Since this algorithm is implemented with multi-threading, we only compare it to our non-pipeline method to study its effects. In our

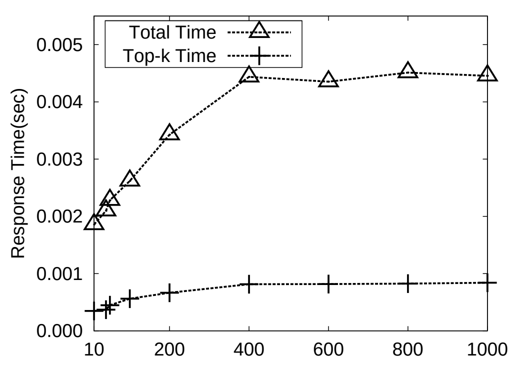


(a) Response time vs. τ

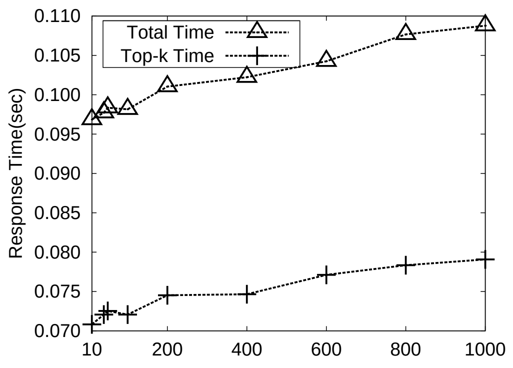


(b) Access ratio vs. τ

Figure 3.19: Quality of **SEGOS**



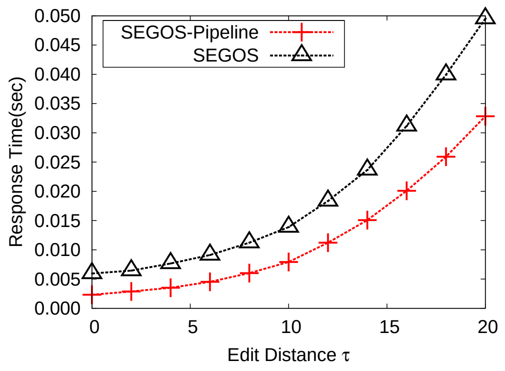
(a) On AIDS dataset



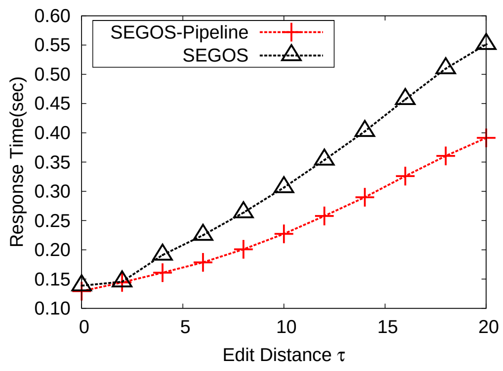
(b) On Linux dataset

Figure 3.20: Overhead testing of top-k sub-unit search on range queries

implementation, we dispatch two parallel threads to respectively run the TA and the CA stage, and two threads to run the DC stage respectively for two parallel parts: partial matching computations and sub-unit multiset difference computations. With this, the overhead of **SEGOS** can be reduced by parallel processing. **SEGOS-Pipeline** fixes the k_s value to be 20, and CA feeds its output into the DC stage when it finishes processing sorted lists constructed by top-20 results from TA. Therefore, the h parameter can be removed. Figure 4.8 shows one group of results on range queries, varying τ from 0 to 20. In this experiment, we randomly select 20K data graphs and 20 query graphs from two real datasets. The results show that the pipelined algorithm can further speed up the graph search. The access number for queries does not exceed 700, which is not significant enough to show the high enhancement. However, the trend shows that with τ increasing, the enhancement becomes higher and higher.



(a) On AIDS dataset



(b) On Linux dataset

Figure 3.21: Effects of pipeline on **SEGOS**

3.6 Summary

In this study, we investigate an important problem of GED based graph similarity search. Different from previous works, we propose **SEGOS**, an efficient indexing and pipeline query processing framework based on sub-units. A two-level inverted index is constructed and preprocessed to maintain a global similarity order both for sub-units and graphs. With this blessing property, graphs can be accessed in increasing dissimilarity, and any GED based lower/upper bound can be used as filtering features. With this, two algorithms adapted from TA and CA are seamlessly integrated into the framework to speed up the search, and it is easy to pipeline the proposed framework to process continuous graph pruning. The top-k result in the TA stage is automatically fed into the CA stage, and the accessed sub-units of each graph from the CA stage are output to the DC stage. Experimental results on two real datasets show that the proposed approach outperforms the state-of-the-art works with best filtering power. Although κ-**AT** is fast to answer queries but its loose bound causes it to suffer very poor filtering power. Since GED verification is extremely expensive, it makes sense to sacrifice a few more milliseconds to prune as many candidates as possible. **SEGOS** also can highly improve **C-Star** by avoiding accessing the whole database.

CHAPTER 4

KNN Sequence Search with Approximate n-grams

To extend the graph index mechanism to support efficient sequence similarity queries, we propose a novel pipeline framework. The work first introduces new observations on the properties of gram distance which provide new bounds for sequence edit distance. Then, a two-level inverted index is constructed to support the proposed pipeline search framework. The proposed algorithms exploit new properties and offer new opportunities for improving query performance.

4.1 Overview

Given a query sequence, the goal of KNN sequence search is to find k sequences in the database that are most similar to the query sequence. This work has focused on the KNN search problem based on edit distance.

In the literature, existing algorithms have focused on either approximate searching (e.g., [8, 35, 36, 48, 56, 61, 65]) or KNN similarity search [62, 74, 79]. Although range query has been extensively studied, KNN search remains a challenging issue. Many efforts on answering KNN search utilize the filter-and-refine framework [62, 74, 79]. The main idea is to prune off candidates by utilizing the number of exact matches on a set of n-grams that are generated from the sequences. An n-gram is a contiguous subsequence of a particular sequence (also called q-gram). Although such approaches are effective on short sequence searches, they are less effective if there is a need to process sequences that are longer like a page of text in a book. In this work, we further investigate the KNN search problem from the viewpoint of enhancing efficiency.

In this work, we develop a novel search framework which uses approximate n-grams as the filtering signatures. This allows us to use longer n-grams compared to exact matches which in turn gives more accurate pruning since such matching is less likely to be random. We introduce two novel filtering techniques based on approximate n-grams by relaxing the filtering conditions. To ensure efficiency, we employ several strategies. First, we use a frequency queue (f-queue) to buffer the frequency of the approximate n-grams to support candidate selection. This can help to avoid frequent candidate verification. Second, we develop a novel search strategy by employing the paradigm of the CA method [15]. By using the summation of gram edit distances as the aggregation function, the CA strategy can enhance the KNN search by avoiding access to sequences with high dissimilarity. Third, we design a pipeline framework to support simple parallel processing. These strategies are implemented over a two-level inverted index. In the upper-level index, n-grams that are derived from the sequence database are stored in an inverted file with their references to the original sequences. In the lower-level index, each distinct n-gram from the upper-level is further decomposed into smaller sub-units, and inverted lists are constructed to store the references to the upper-level grams for each sub-unit. Based on the index, the search framework has two steps.

In the first step, given a query sequence and its n-grams, similar n-grams within a range will be quickly returned using the lower-level index. In the second step, the n-grams returned from the lower level can be automatically used as the input to construct the sorted lists in the upper level. With the sorted lists, our proposed filtering strategies are employed to enhance the search procedure. Our contributions in this work are summarized as follows:

- We introduce novel bounds for sequence edit distance based on approximate n-grams. These bounds offer new opportunities for improving pruning effectiveness in sequence matching.

- We propose a novel KNN sequence search framework using several efficient strategies. The f-queue supports our proposed filtering techniques with a sequence buffer for candidate selection. The well-known CA strategy has an excellent property of early termination for scanning the inverted lists, and the pipeline strategy can effectively make use of parallel processing to speed up our search.

- We propose a pipeline search framework based on a two-level inverted index. By adopting a carefully staged processing that starts from searching at the lower-level n-gram index to ending at the upper-level sorted list processing, we are able to find KNN for long sequences in an easily parallelizable manner.
- We conduct a series of experiments to compare our proposed filtering strategies with existing methods. The results show that our proposed filtering techniques have better pruning power, and the new filtering s-trategies can enhance existing filtering techniques.

4.2 Preliminaries

Let Σ be a set of elements, e.g. a finite alphabet of characters in a string database or an infinite set of latitude and longitude in a trajectory database. We use s to denote a sequence in Σ^* of length $|s|$, $s[i]$ to denote the ith element, and $s[i, j]$ to denote a subsequence of s from the ith element to the jth element. The common notations used in this work are listed in the "LIST OF SYMBOLS" in Page xii.

In this work, we employ edit distance as the measure on the dissimilarity between two sequences, which is formalized as follows.

Definition 4.1. *(Sequence Edit Distance)(SED) Given two sequences s_1 and s_2, the edit distance between them, denoted by $\lambda(s_1, s_2)$, is the minimum number of primitive edit operations (i.e., insertion, deletion, and substitution) on s_1 that is necessary for transforming s_1 into s_2.*

We focus on k-nearest neighbor (KNN) search based on the edit distance, following the formal definition as below.

Problem 4.1. *Given a sequence database $D = \{s_1, s_2, ..., s_{|D|}\}$ and a query sequence q, find k sequences $\{a_1, a_2, ..., a_k\}$ in D, which are more similar to q than the other sequences, that is, $\forall s_i \in D \backslash \{a_j (1 \leq j \leq k)\}$, $\lambda(s_i, q) \geq \lambda(a_j, q)$.*

4.2.1 KNN Sequence Search Using n-grams

In this section, we aim to introduce important concepts and principles of sequence similarity search using n-grams which is a common technique exploited in existing studies.

Definition 4.2. *(n-gram) Given a sequence s and a positive integer n, a positional n-gram of s is a pair (i, ng), where ng is a subsequence of length n starting at the i^{th} element, i.e., $ng = s[i, i+n-1]$. The set $G(s, n)$ consists of all n-grams of s, obtained by sliding a window of length n over sequence s. In particular, there are $|s| - n + 1$ n-grams in $G(s, n)$.*

In this work, we ignore the positional information of the n-grams. Such a simplified 5-gram set of a sequence *introduction*, for example, is {*intro*, *ntrod*, *trodu*, *roduc*, *oduct*, *ducti*, *uctio*, *ction*}. The n-gram set is useful in edit distance similarity evaluation, based on the following observation: if a sequence s_2 could be transformed to s_1 by τ primitive edit o perations, s_1 and s_2 must share at least $\phi = (\max\{|s_1|, |s_2|\} - n + 1) - n \times \tau$ common n-grams [56].

Algorithm 4.1: A Simple KNN Sequence Search Algorithm

```
Require: The $n$-gram lists $L_G$ for $q$, and $k$
 1: Initialize a max-heap $H$ using first visited $k$ sequences;
 2: for $L_i \in L_G$ do
 3:   for all unprocessed $s_j \in L_i$ do
 4:     $frequency[s_j]++$;
 5:     $\tau = max\{\lambda_s | s \in H\}$;
 6:     $\phi = max\{|s_j|, |q|\} - n + 1 - n \times \tau$;
 7:     if $frequency[s_j] \geq \phi$ then
 8:       Compute the edit distance $\lambda(s_j, q)$;
 9:       if $\lambda(s_j, q) < \tau$ then
10:         Update and maintain the max-heap $H$;
11:       Mark $s_j$ as a processed sequence;
12: Output the $k$ sequences in $H$;
```

Inverted indexes on the n-grams of the sequences are commonly used, such that references to original locations of the same n-gram are kept in a list structure. Algorithm 4.1 shows a typical threshold-based algorithm using the inverted index on the n-grams as well as an auxiliary heap structure. This algorithm dynamically updates the frequency threshold using the maximum edit distance maintained in a max-heap H (lines 6 - 7). The query performance depends on the efficiencies of two operations, the inverted list scan and the edit distance computation for the candidate verification (lines 3 - 11).

Algorithm 4.1 could be improved by using optimization strategies, such as *length filtering* [74] and *MergeSkip* [62]. The intuition behind *length filtering* is as follow: if two sequences are within an edit distance of τ, their length difference is no larger than τ. Therefore, the inverted list scan is restricted to the sequences within the length constraint. Inverted lists are thus sorted in ascending order of the sequence length. On the other hand, the *MergeSkip* strategy preprocesses inverted lists such that the references are sorted in ascending order of sequence id number. When the maximum entry in the max-heap H is updated, it is used to compute a new frequency threshold ϕ, and those unprocessed sequences with frequencies less than ϕ are skipped. As an example, in Figure 4.1, sequence no. 10 is first visited and pushed to the top-1 heap. The temporal frequency threshold is computed as $\phi = 3$, and the candidate for next visit is sequence no. 35. In this way, sequences 20 and 30 are skipped as their frequencies are less than 3.

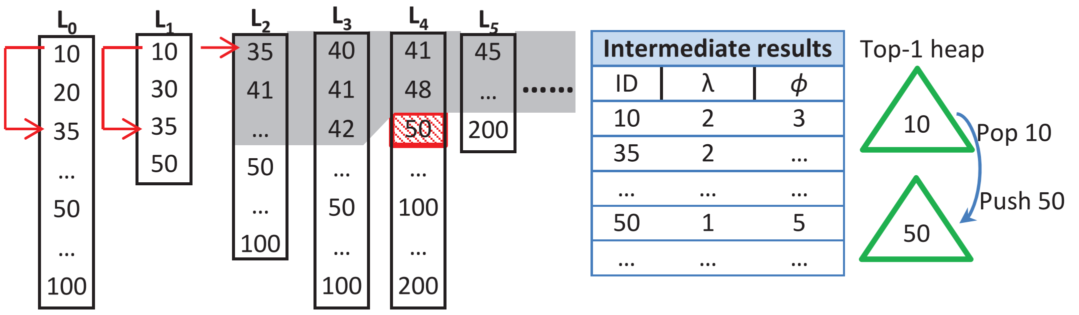


Figure 4.1: Illustration of the MergeSkip strategy

Although such approaches may somehow improve the efficiency of list processing, they may have limited performance since they are strictly relying on the efficient processing of inverted lists. For example, the length filtering can be useless in a database where most sequences are around the same length. In Figure 4.1, the top-1 heap is updated when sequence no. 50 is visited, the new frequency threshold is $\phi = 5$, and the next visiting candidate is sequence no. 45. In this case, no sequence may be skipped. The reason is that sequences from 35 to 45 are located in the grey area may have been processed as the frequencies of their matched n-grams are larger than 3. As the frequency threshold is a loose bound that can generate too many false positives, the candidate verification becomes the most time consuming step.

4.3 New Filtering Theory

Due to the limited pruning effectiveness of exact n-gram matching, we aim to develop new theories for sequence search filtering by using approximate matching between n-grams of the two sequences. This is motivated by the observations that using exact n-gram matching will typically require n to be small (so that the probability of having exact matching will not be too low) which will in turn lower the selectivity of the n-grams. By allowing approximate matching for these n-grams, we can increase the size of n without compromising the chance of a matching taking place, thereby increasing selectivity of the n-grams and reducing the length of the inverted list to be scanned. We will first define what we refer to as *gram edit distance.*

Definition 4.3. *(GRam Edit Distance)(GRED) Given two n-grams ng_1 and ng_2, the edit distance between them, denoted by $\lambda(ng_1, ng_2)$, is the minimum number of primitive edit operations (i.e., insertion, deletion, and substitution) to transform from ng_1 to ng_2.*

Count filtering is the first pruning strategy we design based on gram edit distance, It is an extension of the existing count filtering on exact n-gram matchings. Basically, we want to estimate the maximal number of n-grams modified by τ edit operations such that the gram edit distance between the affected n-gram and the queried n-gram is larger than a certain value of t ($t \geq 0$). This leads to the new count filtering using approximate n-grams, as is shown in the following proposition.

Propositon 4.1. *Consider two sequences s_1 and s_2. If s_1 and s_2 are within an edit distance of τ, then s_1 and s_2 must have at most $\eta(\tau, t, n) = \max\{1, n - 2 \times t\} + (n - t)(\tau - 1)$ n-grams with gram edit distance $> t$, where $t < n$.*

Proof. Let $t = 0$. Then $\eta(\tau, 0, n) = \max\{1, n - 2 \times 0\} + (n - 0) \times (\tau - 1) = n \times \tau$. Intuitively, this holds because one edit operation can modify at most n n-grams. Consequently, τ edit operations can modify at most $n \times \tau$ n-grams (i.e., there are at most $n \times \tau$ n-grams between s_1 and s_2 with gram edit distance > 0).

Let $t \geq 1$. We first analyze the effect of edit operations on the n-grams with certain gram edit distance (GRED). We show the first edit operation in two

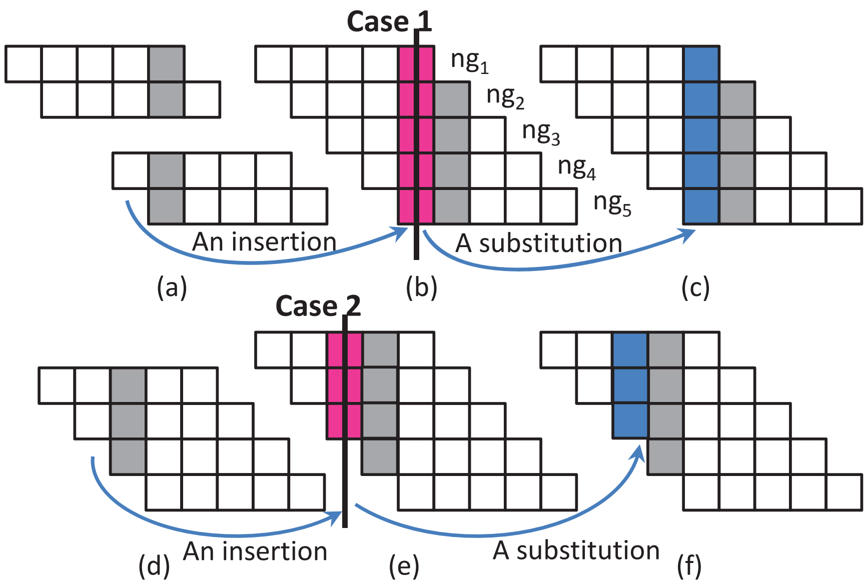
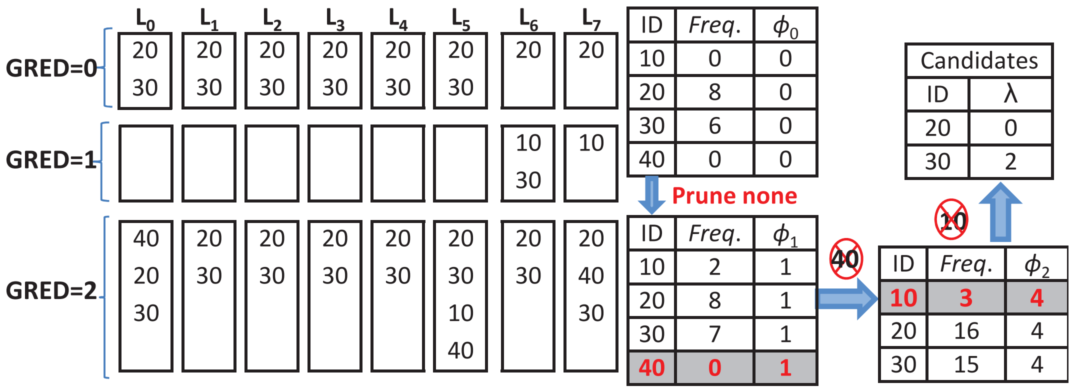


Figure 4.2: Effect of edit operations on n-grams

cases: it is applied on the first or last n-1 n-grams, and it is applied into other positions not within the first or last n-1 n-grams. As shown in Figure 4.2, in Case 1, one edit operation is applied in the position in the pink box. Two types of edit operations will affect n-grams to have different distance distributions. Obviously, one substitution will cause n n-grams to have GRED $= 1$; while one insertion or deletion will cause one new n-gram and n-1 n-grams of various GREDs. Consequently, the upper bound value of $\eta(\tau, t, n)$ will cause at least 1 n-gram with GRED $= n$. We now show the distribution of the GREDs. As shown in the figure, two 5-grams ng_1 and ng_5 have GRED $= 1$ in Figure 4.2(a). However, two 5-grams ng_2 and ng_4 can have GRED ≤ 2. Generally, one such operation can cause at most $n - 2 \times t$ n-grams to have GRED $> t$. Remember that there are at least one new derived n-gram of GRED $= n$. Therefore, an upper bound on the number of affected n-grams with GRED $> t$ should be $\max\{1, n-2 \times t\}$. In case 2, one edit operation is applied to the first or last n-1 n-grams. The total number of affected n-grams, denoted by n', is less than n, and the number of affected n-grams have GRED $> t$ should be less than that of Case 1. It is obvious that the number of affected n-grams in Case 2 is less than that in Case 1. It is indeed true that Case 1 can infer an upper bound value on the affected n-gram number when the insertion or deletion operation is applied.

We now show how the distribution of edit operations will affect the maximum number of n-grams with GRED $> t$. Suppose $E = \{e_1, e_2, \ldots, e_\tau\}$ is a series of edit operations that is needed to transform one sequence into another sequence. Suppose the τ edit operations are evenly distributed in a sequence. It is easy to show that the number of affected n-grams is maximized . As analyzed above, one edit operation will affect at most $n - 2 \times t$ n-grams to have GRED $> t$. This is the boundary case where the edit operation is the first or the last operation. It is clear that the number of affected n-grams with GRED $> t$, on the left of the first edit operation and on the right of the last edit operation, is at most $\max\{1, n - 2 \times t\}$. For the last $\tau - 1$ edit operations, it is easy to show that one new operation will cause n-t newly affected n-grams ahead its previous edit position, as the boundary position will be affected only in this case. Consequently, the maximum number of affected n-grams with GRED $> t$ would be $\eta(\tau, t, n) = \max\{1, n - 2 \times t\} + (n - t)(\tau - 1)$. $\square$

Lemma 4.1. *Consider two sequences s_1 and s_2. If s_1 and s_2 are within an edit distance of τ, then s_1 and s_2 must share at least $\phi_t(s_1, s_2) = |s| - n + 1 - \eta(\tau, t, n)$ n-grams with gram edit distance $\leq t$. Here, $|s|$ is equal to $\max\{|s_1|, |s_2|\}$ and the positional information is ignored.*



ID	Freq.	ϕ_0
10	0	0
20	8	0
30	6	0
40	0	0

ID	Freq.	ϕ_1
10	2	1
20	8	1
30	7	1
40	0	1

ID	Freq.	ϕ_2
10	3	4
20	16	4
30	15	4

Candidates	
ID	λ
20	0
30	2

Figure 4.3: An example of the count filtering

The proposed count filtering offers new opportunities to improve the search performance as it has a stronger filtering ability. As is shown in Figure 4.3, no sequence is pruned using the count filtering with common n-grams of $\phi_0 = 0$. By using the count filtering with n-grams of GRED $= 1$, sequence no. 40 can be pruned by ϕ_1 as its frequency (i.e., *Freq.*) of n-grams with GRED ≤ 1 is less than ϕ_1. Similarly, the sequence 10 is pruned by using the count filtering of ϕ_2.

Mapping filtering is a more complicated pruning strategy, but provides more effective pruning based on the gram edit distance. To begin with, we first define the distance between two multi-sets of n-grams.

Definition 4.4. *(GRam Mapping Distance)(GRMD) Given two gram multi-sets G_{s_1} and G_{s_2} of s_1 and s_2, respectively with the same cardinality. The mapping distance between s_1 and s_2 is defined as the sum of distances of the optimal mapping between their gram multi-sets, and is computed as*

$$\mu(s_1, s_2) = \min_P \sum_{ng_i \in G_{s_1}} \lambda(ng_i, P(ng_i)), \ \ P : G_{s_1} \to G_{s_2}$$

The computation of gram mapping distance is accomplished by finding an optimal mapping between two grams multi-sets. Similar to the work in [67], we can construct a weighted matrix for each pair of grams from two sequences, and apply the Hungarian algorithm [33, 60]. Based on gram mapping distance, we show how a tighter lower bound on the edit distance between two sequences could be achieved.

Lemma 4.2. *Given two sequences s_1 and s_2. The gram mapping distance $\mu(s_1, s_2)$ between s_1 and s_2 satisfies*

$$\mu(s_1, s_2) \leq (3n - 2) \cdot \lambda(s_1, s_2)$$

Proof. Let $E = \{e_1, e_2, \ldots, e_K\}$ be a series of edit operations that is needed to transform s_1 into s_2. Accordingly, there is a set of sequences $s_1 = M_0 \to M_1 \to \ldots \to M_\tau = s_2$, where $M_{i-1} \to M_i$ indicates that M_i is the derived sequence from M_{i-1} by performing e_i for $1 \leq i \leq K$. Assume there are K_1 insertion operations, K_2 deletion operations and K_3 substitution operations, then we have $K = K_1 + K_2 + K_3$. We analyze the detailed influence of each type of edit operation as follows.

Insertion operation: When a character is inserted into the sequence M_{i-1}, at most n n-grams are affected. The edit distance is less than 2 for $(n-1)$ n-grams , and n for one newly inserted n-gram. Thus, we conclude that $\mu(M_{i-1}, M_i) \leq [2(n-1) + n] = 3n - 2$.

Deletion operation: When one character is deleted from the sequence M_{i-1}, thus a total number of n n-grams may be affected. The edit distance is less than 2 for $(n-1)$ n-grams, and n for one newly deleted n-gram. Thus, in the case of deleting one character, $\mu(M_{i-1}, M_i) \le [2(n-1)+n] = 3n-2$.

Substitution operation: When a character in sequence M_{i-1} is substituted by another character, a total number of n n-grams are affected. Then, the edit distance for each affected n-gram is equal to 1, and thus we have $\mu(M_{i-1}, M_i) \le n$.

By analyzing the effect of the above three operations, we conclude that GRMD and SED have the following relationship.

$$\begin{aligned}
\mu(s_1, s_2) &\le (3n-2)\cdot K_1 + (3n-2)\cdot K_2 + n\cdot K_3 \\
&\le (3n-2)\cdot(K_1+K_2+K_3) \\
&\le (3n-2)\cdot\lambda(s_1, s_2)
\end{aligned}$$

□

Lemma 4.2 naturally brings us a new lower bound estimation method on the sequence edit distance. Given two sequences s_1 and s_2, and an edit distance threshold τ, if $\frac{\mu(s_1,s_2)}{3n-2} > \tau$, then $\lambda(s_1, s_2) > \tau$. While the bound is effective, it remains computational expensive if we directly apply the pruning theories presented in this section. In this work, we employ this bound function to compute the aggregation value in the CA filtering algorithms. That is, we use the summation of gram edit distances as the aggregation function, instead of directly computing the mapping distance. We will introduce new implementing filtering strategies and algorithmic frameworks to make these theories practical.

4.4 Filtering Algorithms

Based on the filtering theories derived in the previous section, we introduce new algorithms to support efficient filtering. Given a query sequence q, we assume that there are existing inverted lists that support efficient search on the n-grams under specified edit distance constraint, as shown in Figure 4.4 and 4.5 with $L_G = \{L_0, L_1, ..., L_{|q|-n}\}$.

We use a frequency queue (f-queue) to speed up our inverted list processing. The basic intuition is that sequences that share higher number of matched approximate n-grams with the query sequence will be given preference for processing. The f-queue is an unordered queue, which maintains the top-k' unique visited sequences with frequency of approximately matched n-grams larger than a temporary frequency threshold.

Algorithm 4.2: KNN Search Algorithm Using the F-queue

```
Require: The $n$-gram lists $L_G$ with GRED $= t$ for $q$, and $k'$
 1: Initialize the f-queue as $\emptyset$;
 2: Initialize a max-heap $H$ using first visited $k$ sequences;
 3: for $L_i \in L_G$ do
 4:   for all unprocessed $s_j \in L_i$ do
 5:     $frequency[s_j]++$;
 6:     Update the top-$k'$ f-queue;
 7:     if $k'$ items in f-queue then
 8:       for all $s_c \in top-k'$ do
 9:         $\tau = max\{\lambda_s | s \in H\}$;
10:         $\eta(\tau, t, n) = \max\{1, n - 2 \times t\} + (n - t)(\tau - 1)$;
11:         $\phi_t = max\{|s_c|, |q|\} - n + 1 - \eta(\tau, t, n)$;
12:         if $frequency[s_c] \geq \phi_t$ then
13:           Compute the edit distance $\lambda(s_c, q)$;
14:           Mark $s_c$ as a processed sequence;
15:           if $\lambda(s_c, q) < \tau$ then
16:             Update and maintain the max-heap $H$;
17: Output the $k$ sequences in $H$;
```

The f-queue is first initialized to be an empty set, and we perform access to the inverted lists to count the frequency of approximately matched n-grams. As shown in Algorithm 4.2, if the queue contains k' unprocessed sequences, our algorithm first sorts the visited sequences in ascending order based on the frequencies. Subsequently, we verify the sequences in the f-queue using the temporary frequency threshold (highest frequency first). Those sequences passing the count filtering are verified with the exact edit distance computation, and used to update the top-k heap. Note that the temporary frequency threshold is immediately updated when a new value is inserted into the top-k heap. Generally, the f-queue technique avoids frequent verifications on the visited sequences. It offers new opportunities to employ count filtering with approximate n-grams. The f-queue can be used to improve the performance of existing algorithms based on the length filtering or the MergeSkip strategy.

Figure 4.4 illustrates the idea of the f-queue and explains why it improves

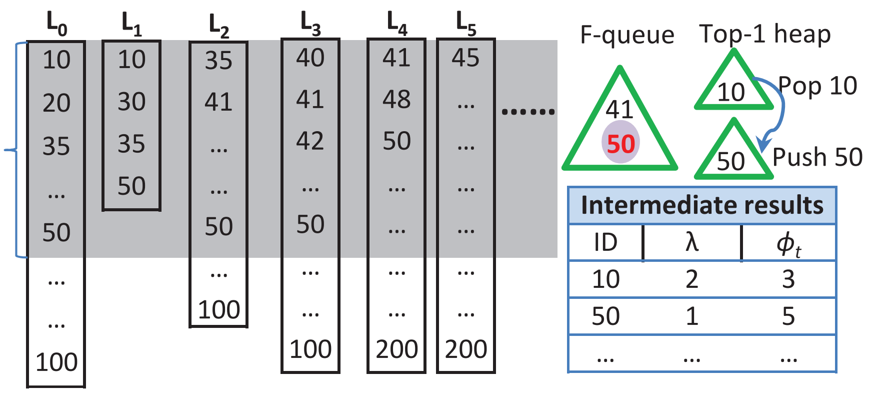


Figure 4.4: Illustration of the frequency queue

the *MergeSkip* strategy. The top-1 heap is initialized using the first sequence which is visited i.e. sequence no. 10 and the frequency threshold is set as 3. Here, we set the value of k' as 2. After scanning the lists in the gray box, two unprocessed sequences no. 41 and 50 are pushed into the f-queue since the frequency of their approximately matched n-grams are higher than the temporary frequency threshold. The f-queue is then traversed for candidate verification, and sequence no. 50 is verified first since it possesses a highest number of matching n-grams. As its edit distance is computed as 1, a new threshold is computed for the top-1 matching and the top-1 heap is updated accordingly, by discarding the sequence no. 10 and pushing sequence no. 50. Finally, the new frequency threshold is 5, based on our update rule. Compared against the standard method in Figure 4.1, our approach successfully skips the sequences from 35 to 45.

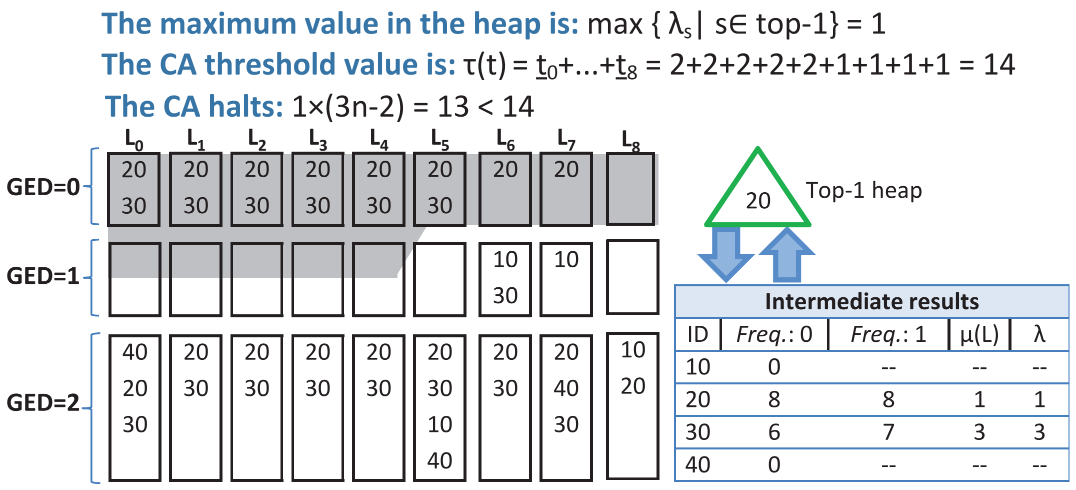
As the novel count filtering can be applied without any constraint on the gram edit distance, the list processing with the f-queue can continue until all the sequences are processed. However, applying more count filters will mean that more list processing time is required. To avoid having large overhead on list processing, we use the CA based strategy [15] and use the summation of gram edit distances as the aggregation function. As inverted lists of certain GRED are fetched out separately, they are naturally sorted by the gram edit distance (lowest distance first). Algorithm 4.3 shows the details of the CA based filtering algorithm. We use Example 4.1 to illustrate the effectiveness of the CA-based filtering framework for supporting the KNN search.

Example 4.1. *In Figure 4.5, we consider the nine sorted lists for a query*

Algorithm 4.3: KNN Search Algorithm Using the CA Method

```
Require: The $n$-gram lists $L_G$ with GRED $= t$ for $q$
1: Initialize the f-queue as $\emptyset$;
2: Initialize a max-heap $H$ using first visited $k$ sequences;
3: Initialize $\underline{t_i} = t$ with $i = 0 \ldots |q| - n$;
4: for $L_i \in L_G$ do
5:    for all unprocessed $s_j \in L_i$ do
6:       $frequency[s_j] + +$;
7:       Update the top-$k'$ f-queue;
8:       if The end of $L_i$ is visited then
9:          $\underline{t_i} = t + 1$;
10:         if $t > 0$ then
11:            $\tau = max\{\lambda_s | s \in H\}$;
12:            $\tau(t) = \sum_{l=0}^{l=|q|-n} \underline{t_l}$;
13:            if $\tau(t) \geq \tau \times (3n - 2)$ then
14:               Terminate the list processing;
15:      if $k'$ items in f-queue then
16:         Apply the filtering strategy with the f-heap;
17: Output the $k$ sequences in $H$;
```

sequence q. The length of the n-gram is set to be $n = 5$. Each list has three groups of n-grams with various GREDs of 0, 1, and 2. Each entry in the lists stores the sequence id number. Suppose we perform sorted access to each sorted list L_i. For each list L_i, let $\underline{t_i}$ be the GRED score of the last sequence visited under sorted access. The CA threshold value is computed as $\tau(t) = \sum_{i=0}^{i=8} \underline{t_i}$. As soon as $\tau(t) \geq max\{\lambda_s | s \in top\text{-}k\} \times (3n - 2)$, CA halts and we stop scanning the inverted lists. In the figure, CA halts at the positions at the bottom of the grey area. In this case, each $\underline{t_i}$ in the group of GRED $= 1$ is initialized to be 1. When we do sorted access to the list L_4, each value of $\underline{t_i}$ with $i = 0, 1, .., 4$ is set to be equal to 2 as no entry has distance of 1, and 2 is the smallest score that can be obtained for unseen elements. Therefore, $\tau(t)$ is computed as $\sum_{i=0}^{i=8} \underline{t_i} = 2 + 2 + 2 + 2 + 2 + 1 + 1 + 1 + 1 = 14$. Consequently, the CA halts as $\tau(t) \geq max\{\lambda_s | s \in top\text{-}1\} \times (3n - 2) = 1 \times 13$. According to Lemma 4.2, the unseen sequences 10 and 40 are safely pruned.



The maximum value in the heap is: max { λ_s | s∈ top-1} = 1

The CA threshold value is: τ(t) = $\underline{t}_0$+...+$\underline{t}_8$ = 2+2+2+2+2+1+1+1+1 = 14

The CA halts: 1×(3n-2) = 13 < 14

Intermediate results				
ID	*Freq.*: 0	*Freq.*: 1	μ(L)	λ
10	0	--	--	--
20	8	8	1	1
30	6	7	3	3
40	0	--	--	--

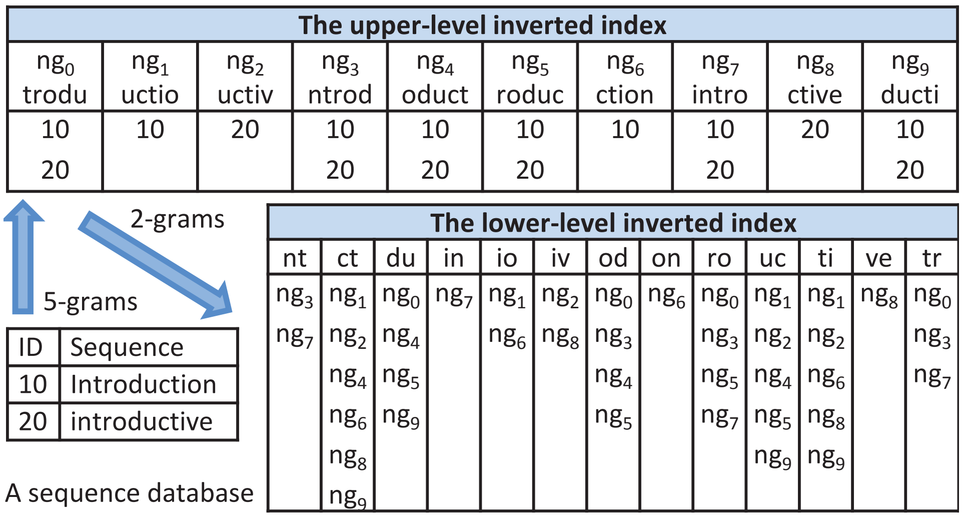
Figure 4.5: An example of CA based filtering

4.5 Indexing and Query Processing

In the previous section, we developed our algorithms based on the assumption that we can find the approximately matched n-grams efficiently and is thus able to access the corresponding inverted list of these approximately matched n-grams. In this section, we will explain how this can be done on top of a two-levels inverted index. Based on this two-levels index, we will develop a pipeline framework to support efficient KNN sequence search.

We build a two-levels inverted index based on n-grams with different granularity of n_1 and n_2 respectively. As shown in Figure 4.6, the index consists of the upper-level index and the lower-level index. In particular, given a sequence database D, the upper-level is used to index the n_1-grams that are obtained from the original sequences in D, and the lower-level is used to index the n_2-grams that are obtained from n_1-grams in the upper-level ($n_1 > n_2$).

There are two steps to build the index: 1) we extract n_1-grams from sequences in D, and build the upper-level inverted index. The index is made up of two main components: an index for all distinct n_1-grams and an inverted list below each n_1-gram. In general, each entry in the inverted lists contains the sequence identifier. 2) we further extract n_2-grams from all distinct n_1-grams, and build the lower-level inverted index. Similarly, the index consists of two main parts, which stores the n_2-grams with the reference to the corresponding n_1-gram in the inverted lists. Generally, the inverted list entries in the upper-level index are usually sorted into various orders when using different filtering



The upper-level inverted index

ng_0 trodu	ng_1 uctio	ng_2 uctiv	ng_3 ntrod	ng_4 oduct	ng_5 roduc	ng_6 ction	ng_7 intro	ng_8 ctive	ng_9 ducti
10	10	20	10	10	10	10	10	20	10
20			20	20	20		20		20

The lower-level inverted index

nt	ct	du	in	io	iv	od	on	ro	uc	ti	ve	tr
ng_3	ng_1	ng_0	ng_7	ng_1	ng_2	ng_0	ng_6	ng_0	ng_1	ng_1	ng_8	ng_0
ng_7	ng_2	ng_4		ng_6	ng_8	ng_3		ng_3	ng_2	ng_2		ng_3
	ng_4	ng_5				ng_4		ng_5	ng_4	ng_6		ng_7
	ng_6	ng_9				ng_5		ng_7	ng_5	ng_8		
	ng_8								ng_9	ng_9		
	ng_9											

ID	Sequence
10	Introduction
20	introductive

Figure 4.6: An example of a two-level inverted index in a string database

techniques. For example, they are sorted into order of increasing sequence identifier for the MergeSkip strategy and increasing sequence length for the length filtering. This paper will investigate the effect of the list order on the proposed techniques in the experimental study.

4.5.1 A Simple Serial Solution

We first introduce a simple serial solution using the proposed two-level inverted index. Our approach follows a filter-and-refine framework. Given a query sequence q, it is first decomposed into a set of n_1-grams G_q. As shown in Figure 4.7, the serial algorithm works as follows.

1. **GS**: For each $ng_i \in G_q$, the lower-level index is used to support the sequence similarity search to return n_1-grams with GRED $\leq t$ to ng_i. The returned list of n_1-grams are naturally grouped based on the GRED distance of $0, 1, ..., t$.

2. **DF**: Given the output from the GS step, we fetch out the inverted lists from the upper-level index for those matching n_1-grams of distance 0. The list merging algorithm with the proposed f-queue technique is employed to support fast frequency aggregation and maintain the top-k queue for processed sequences (See Algorithm 4.2 in Section 4.4).

3. **CA**: If the DF step does not halt the algorithm, we further fetch the inverted lists from the upper-level index for those similar n_1-grams of distance t ($t \geq 1$). Given the f-queue and the top-k queue from the output of the DF step, the list merging algorithm will continue to accumulate frequencies of similar n_1-grams, and use the proposed count filtering bound for further pruning. Noted that, the CA filtering bound will be employed if a new gram edit distance appears (See Algorithm 4.3 in Section 4.4).

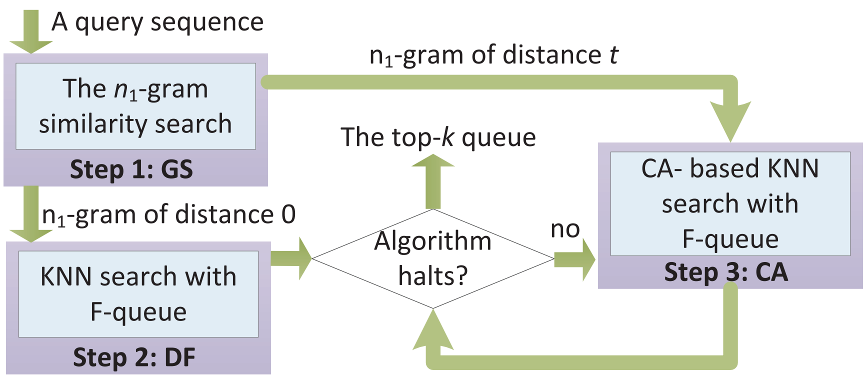


Figure 4.7: The simple serial query processing flow

In the GS step, we employ the fastest approximate string matching algorithm to support efficient n_1-gram similarity search. In our implementation, we use the traditional count filtering with exact n-gram matching to do pre-pruning. The quality of the approximate string matching is sufficiently high to return similar n_1-grams with negligible query time.

We employ the proposed f-queue technique in the DF step and adopt a CA based algorithm to do further pruning in the CA step if the previous processing cannot terminate our search. The algorithm halts under the following conditions: 1) All sequences in the database have gone through the candidate verification; 2) The maximum value in the root of the top-k queue is within an acceptable small distance. Generally, if the temporary frequency threshold is equal to 0, those unseen sequences cannot be safely pruned and need post processing. The algorithm will request for further list processing by relaxing the distance threshold for n_1-gram if these conditions are not met.

In general, the filter-and-refine framework can be evaluated using the cost model with $T_q = T_f + |C_q| \times T_r$, where T_f is the filtering time and T_r is the verification time for each candidate. Our solution attempts to reduce the number of false positives, as it is costly to verify the candidates for long sequences. Compared with existing n-gram based methods, our approach offers new opportunities to speed up the query processing in the CA step by using our novel count filtering.

4.5.2 A Novel Pipeline Framework

Next, we propose a dynamic method that is easy to be pipelined to enhance query processing. The main idea is that similar n_1-grams are dynamically returned from the lower-level index for pruning in the upper-level index.

As shown in Figure 4.8, we adapt the CA step to perform the frequency accumulation without processing sequences using the f-queue. We develop a pipelined algorithm to execute the GS step and the CA step. In this way, the DF step can process visited sequences in the f-queue by using the temporary frequency thresholds that is computed from the approximate n_1-grams, instead of only using the frequency bound of common n_1-grams. The pipelined execution offers new opportunities for reducing the overhead costs of employing multiple filtering techniques.

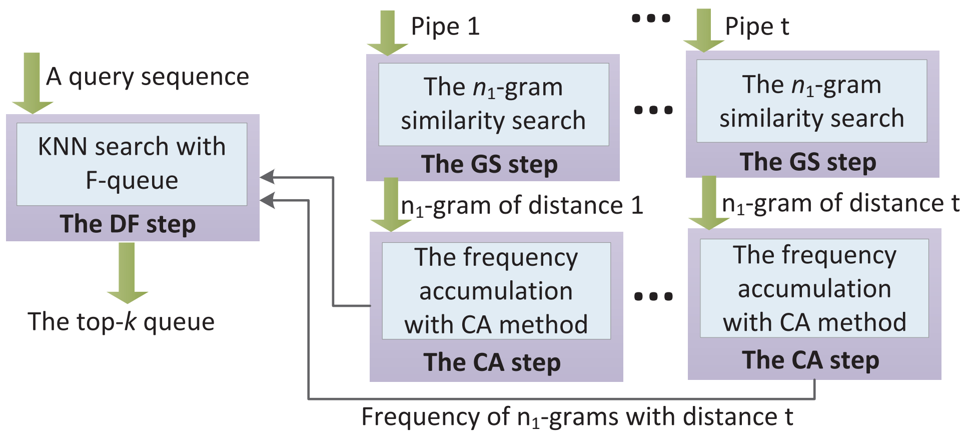

A query sequence
KNN search with F-queue
The DF step
The top-k queue
Pipe 1
The n_1-gram similarity search
The GS step
n_1-gram of distance 1
The frequency accumulation with CA method
The CA step
Pipe t
The n_1-gram similarity search
The GS step
n_1-gram of distance t
The frequency accumulation with CA method
The CA step
Frequency of n_1-grams with distance t

Figure 4.8: The pipelined query processing flow

4.5.3 The Pipelined KNN Search

To support efficient KNN search, the three stages are implemented differently in the pipeline framework as described below.

The GS stage are shown in Algorithm 4.4. The algorithm takes the n_1-gram set G_q obtained from a query q as the input. Given a constraint on the maximum value of gram edit distance (GRED) value, this stage performs the similarity search to return n_1-grams with GRED $= t$ to each $ng_i \in G_q$. The output of the query results will be fed into the CA stage.

Algorithm 4.4: The GS stage

```
Require: The $n_1$-gram set $G_q$, a constraint of $t_{max}$
 1: loop
 2:   Update the GRED threshold value of $t$;
 3:   if The halting signal is detected then
 4:     Terminate this stage;
 5:   else
 6:     for $L_i \in G_q$ do
 7:       Apply $gramSimilaritySearch(ng_i, t)$;
 8:     Pipe similar $n_1$-grams with GRED $= t$ to the CA stage;
```

Algorithm 4.5: The CA stage

```
Require: The global top-$k$ heap $H$
 1: loop
 2:   Update the GRED threshold value of $t$;
 3:   if The halting signal is detected then
 4:     Terminate this stage;
 5:   Obtain inverted lists $L_G$ from the upper-level index for all $n_1$-grams;
 6:   for $L_i \in L_G$ do
 7:     for all unprocessed $s_j \in L_i$ do
 8:       $frequency[s_j]++$;
 9:       if The end of $L_i$ is visited then
10:         $\underline{t_i} = t + 1$;
11:         if $t > 0$ then
12:           $\tau = max\{\lambda_s | s \in H\}$;
13:           $\tau(t) = \sum_{l=0}^{l=|q|-n} \underline{t_l}$;
14:           if $\tau(t) \geq \tau \times (3n - 2)$ then
15:             All unseen strings are safely pruned;
16:             Send a global halting signal;
17:             Terminate this stage;
```

In the CA stage, given the n_1-gram set with GRED $= t$, the inverted lists are fetched from the upper-level index, and scanned to accumulate the frequencies for each visited sequence. The CA strategy is used to terminate the whole process if the CA threshold value of the gram edit distance summation is larger than the temporary threshold computed from the top-k heap. It is convenient to compute the new CA threshold value $\tau(t)$ with a summation of the gram edit distance returned from the GS stage. For example, if the GRED for all returned grams is equal to t, then we have $\tau(t) = t \times |G_q|$. However, this value is updated when a new distance value appears. As shown in Line 9 and Line 12 in Algorithm 4.5, CA can enhance the total query processing by avoiding access to those very dissimilar strings. If the halting condition with the CA aggregation value has been found, this stage immediately stops and sends a global signal to invoke the termination of the whole search. The details of the CA stage are shown in Algorithm 4.5.

In the DF stage, we maintain a global max-heap H for storing the current top-k similar sequences and use the maximum edit distance score in the root of the heap to update the temporary frequency value. As shown in Algorithm 4.6 (lines 13 - 15), the distance value of the top element in the top-k heap is selected as a new range bound for the CA stage and the DF stage.

4.6 Experimental study

We compare the performance of our proposed approach **AppGram** against several state-of-the-art methods over a wide spectrum of real datasets.

4.6.1 Setup

The algorithms used in the following experiments are presented as below.

- B^{ed}**-tree** [79] is proposed to support the string similarity queries using a B^{+}-tree based index structure. We use the implementation from the authors.
- **Flamingo** [35] is an open-source data cleaning system which supports approximate string search. We use the latest release 4.1. For existing work [62] and [74], we use the implementation from the Flamingo, as it has integrated the previous filtering techniques.
- **TopkSearch** [14] is the most recent method that is proposed to support top-k sequences similarity search with edit-distance constraints. We

Algorithm 4.6: The DF stage

```
Require: A query sequence $q$
Require: The global top-$k$ heap $H$
 1: Initialize the f-queue as $\emptyset$;
 2: Initialize a max-heap $H$ using first visited $k$ sequences;
 3: Obtain the $n_1$-gram set $G_q$ from $q$
 4: Obtain $n$-gram lists $L_G$ for all $ng_i \in G_q$
 5: for $L_i \in L_G$ do
 6:     for all unprocessed $s_j \in L_i$ do
 7:         if The halting signal is detected then
 8:             Terminate this stage;
 9:         $frequency[0][s_j] + +$;
10:         Update the top-$k'$ f-queue;
11:         if $k'$ items in f-queue then
12:             for all $s_c \in top - k'$ do
13:                 $\tau = max\{\lambda_s | s \in H\}$;
14:                 $\eta(\tau, t, n) = \max\{1, n - 2 \times t\} + (n - t)(\tau - 1)$;
15:                 $\phi_t = max\{|s_c|, |q|\} - n + 1 - \eta(\tau, t, n)$;
16:                 for all $l = 0 \ldots t$ do
17:                     $frequency(s_c)_l = \sum_{m=0}^{m=l} frequency[m][s_c]$;
18:                 if all $frequency(s_c)_l \geq \phi_l$ with $l = 0 \ldots t$ then
19:                     Compute the edit distance $\lambda(s_c, q)$;
20:                     Mark $s_c$ as a processed sequence;
21:                     if $\lambda(s_c, q) < \tau$ then
22:                         Update and maintain the max-heap $H$;
23: Output the $k$ sequences in $H$;
```

obtain the executable binary file from the authors.

We use two real datasets that are available publicly. They cover different domains and are widely used in previous studies. We use another paragraph dataset obtained from an ebook reading system. The details of the datasets are shown as follows, and the statistics are shown in Table 4.1.

- **IMDB** consists of movie titles which are taken from a public database of IMDB[1], and we use the dataset provided in paper [79].

- **DBLP** consists author names and titles of publications which are extracted from the DBLP Bibliography[2], and we use the dataset provided in paper [65].

[1]http://www.imdb.com
[2]http://www.informatik.uni-trier.de/~ley/db

- **ANNOTEXT** is a dataset containing annotated paragraphs. We extract the paragraphs from a paper abstract collection that are taken from a public citation database[3]. The annotated paragraphs are generated with a random sampling method. We maintain the length distribution of this dataset to be the same as the DBLP dataset.

Table 4.1: Sequence datasets

Dataset	**Size**	**Avg. Len**	**Max. Len**
IMDB	1,553,914	19	240
DBLP	1,385,668	105	1626
ANNOTEXT	1,572,561	75	1250

The query files of the first two public datasets are also available in their original work. Each query file includes 100 sequences, and we obtain them from the authors together with the datasets. For the ANNOTEXT dataset, we select 100 queries by random sampling. Table 4.2 presents major parameters used in our experiments, including their descriptions and values (with default values in bold). Hereafter, the default values will be used in all the experiments unless otherwise stated.

Table 4.2: Parameter settings on KNN sequence search

Parameter	**Description**	**Value**
k	k value for the search	$1, 2, \mathbf{4}, 6, 8, 10$
$\|q\|$	average query size	$10, 20, 30, 40, 50$
τ	distance threshold	$1, 2, 4, 8, 10, 16$

AppGram was implemented in C++, and the pipeline algorithm is implemented using pthread. In all the experiments, we only implement two threads to support two pipelines. We compiled all the algorithms with gcc 4.4.6 in Red hat Linux Operating System, and all experiments were done on a server with Quad-Core AMD Opteron(tm) Processor 8356, 128GB memory, running RHEL 4.7AS.

4.6.2 Construction Time and Index Size

Table 4.3 and 4.4 show the construction time and the index size on the three datasets. The n-gram length is set at n=5 for all datasets. In AppGram, the

[3]http://arnetminer.org/citation

Table 4.3: Construction time (sec)

	AppGram	B^{ed}-tree	Flamingo
IMDB	13	57	25.59
DBLP	154.3	35	116.22
ANNOTEXT	89.3	45	64.12

Table 4.4: Index size (MB)

	AppGram	B^{ed}-tree	Flamingo
IMDB	108	63.2	159
DBLP	563	222.9	608
ANNOTEXT	444	183.7	492

gram length for the lower level is set at 3. As shown in the figure, the B^{ed}-tree takes less time and smaller space than AppGram and Flamingo. Since the AppGram decomposes the sequences into n-grams without any prefix and suffix, it takes slightly smaller space than the Flamingo. The AppGram takes slightly more construction time than the Flamingo, as it needs to build the lower-level index for the n-grams in the upper level. As the binary file for the TopkSearch algorithm does not provide the preprocessing time and space costs, we exclude the results on this method.

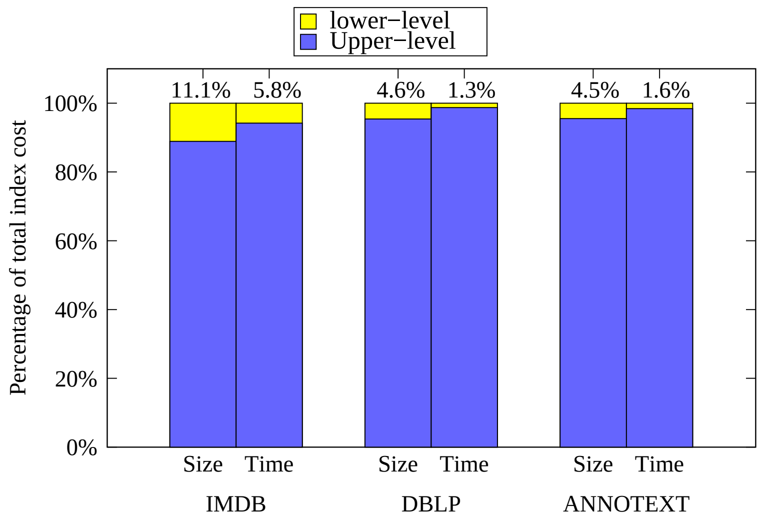
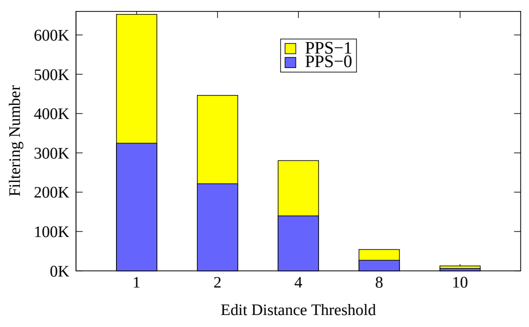


Figure 4.9: Percentage of the index cost

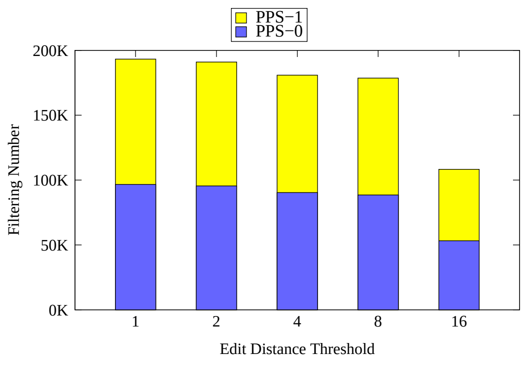
To evaluate the space and time overhead on the extra lower-level index, we present the percentage of total index cost on three datasets in Figure 4.9. Obviously, the lower-level index has a very small ratio compared with the upper-level index. As shown in the figure, it represents 11.1% of the index size over the IMDB dataset, and no more than 5% of the index size over the DBLP and ANNOTEXT datasets. This small cost indicates that the overhead for the lower-level processing with queries may be negligible compared to the total time cost. We will present more results in the following subsections.

4.6.3 Quality of Count Filtering

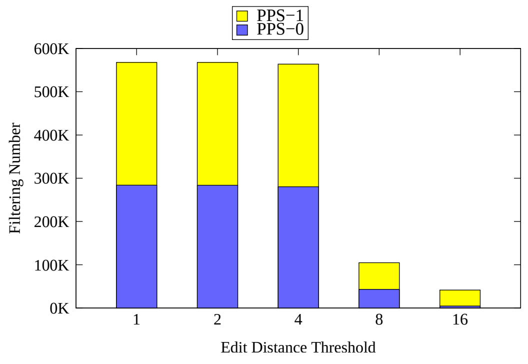
We evaluate the quality of the proposed count filtering technique. The AppGram-0 is the proposed algorithm which employs the common count bound with n-grams of distance 0; while the AppGram-1 is one which uses not only the common count bound but also the count filter with approximate n-grams of distance 1. A query file of 10000 sequences are randomly sampled from each dataset by reserving the original distribution. We vary the edit distance threshold τ as 1, 2, 4, 8, 10, and 16. We accumulate number of sequences which are pruned.



(a) IMDB



(b) DBLP

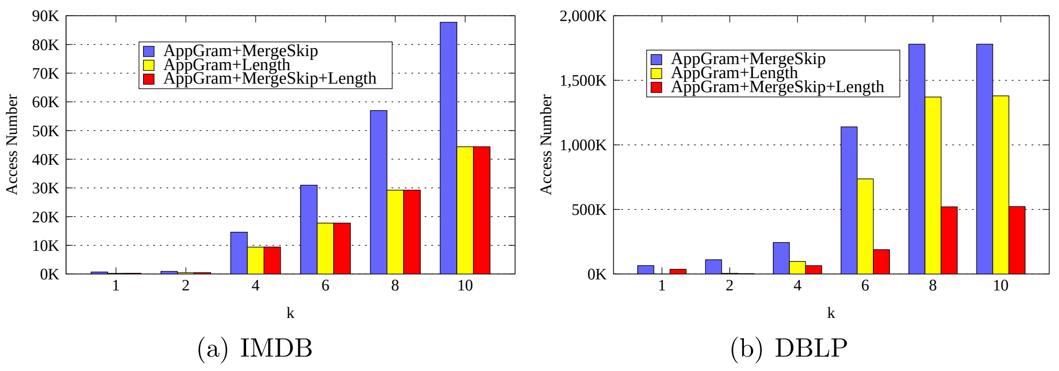


(c) ANNOTEXT

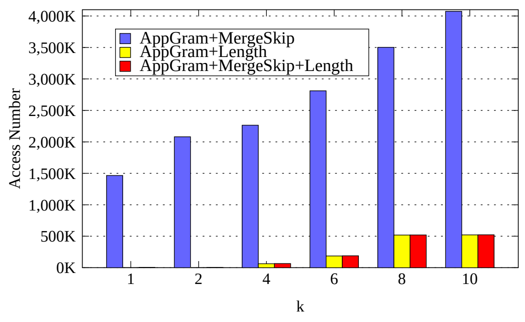
Figure 4.10: Average filtering number vs. τ

Figure 4.10 shows the average number of sequences that are filtered with respect to the edit distance threshold on the three datasets. As shown in the figure, the AppGram-1 can filter out more sequences than the AppGram-0, which means that the proposed count filtering based on approximate n-grams have better filtering power than the existing common count bound.

Generally, the filtering power of the AppGram-1 prunes about 10% more sequences compared with the AppGram-0, and the improvement becomes more significant when the edit distance threshold becomes larger. As shown in Figure 4.10(c), when the edit distance threshold becomes 8 and 16, the AppGram-1 can prune $20K$ sequences compared to $2K$ sequences that are pruned by the AppGram-0. The difference even becomes larger if we further increase the edit distance threshold value.



(a) IMDB (b) DBLP



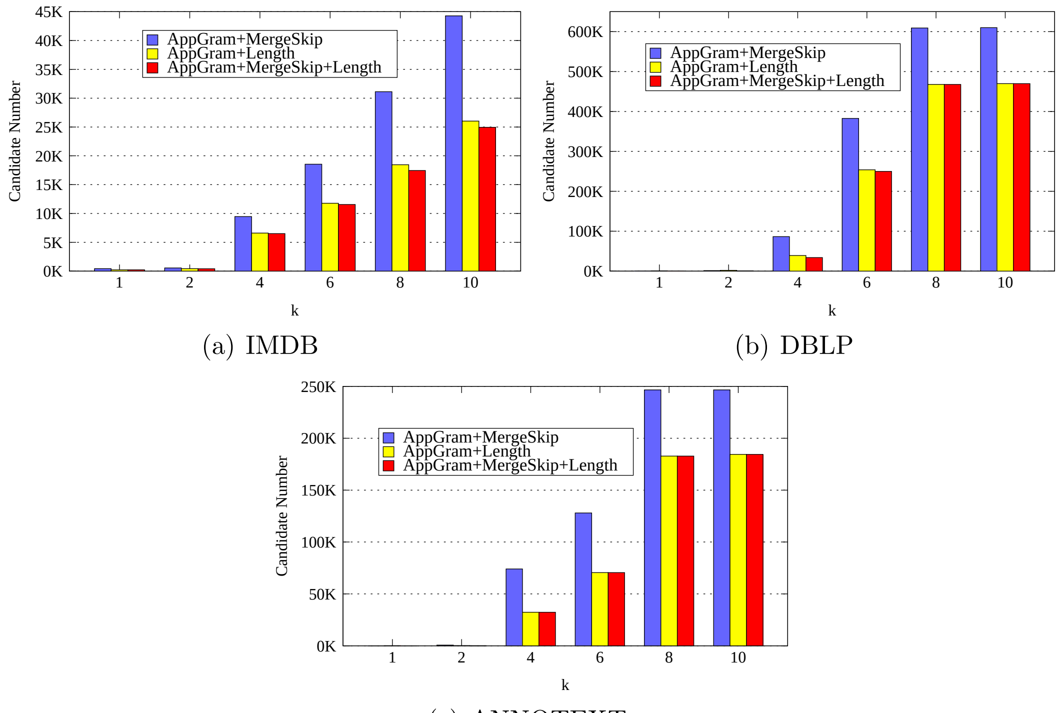
(c) ANNOTEXT

Figure 4.11: Average accessed number of sequences on lists vs. k

4.6.4 Effect of Various Filters

We evaluated various ways to integrate our proposed filters with existing techniques on the three datasets. We use the default query file containing 100 sequences. For each query, we execute the KNN search with various k values from 1 to 10, and count the number of accessed elements in the list processing and the generated candidate size.

Figure 4.11 shows the average number of sequences on the query inverted lists for various filter combinations. As shown in the figure, using only the MergeSkip technique to support dynamic count filtering will access too many sequence ids for processing the inverted lists. The length filtering technique is useful in reducing the number of entries that are accessed for list processing. We also design an algorithm to combine our proposed technique with the MergeSkip technique and the length filter. It is obvious that, the combined technique can significantly reduce the number of accessed entries in the processed lists.

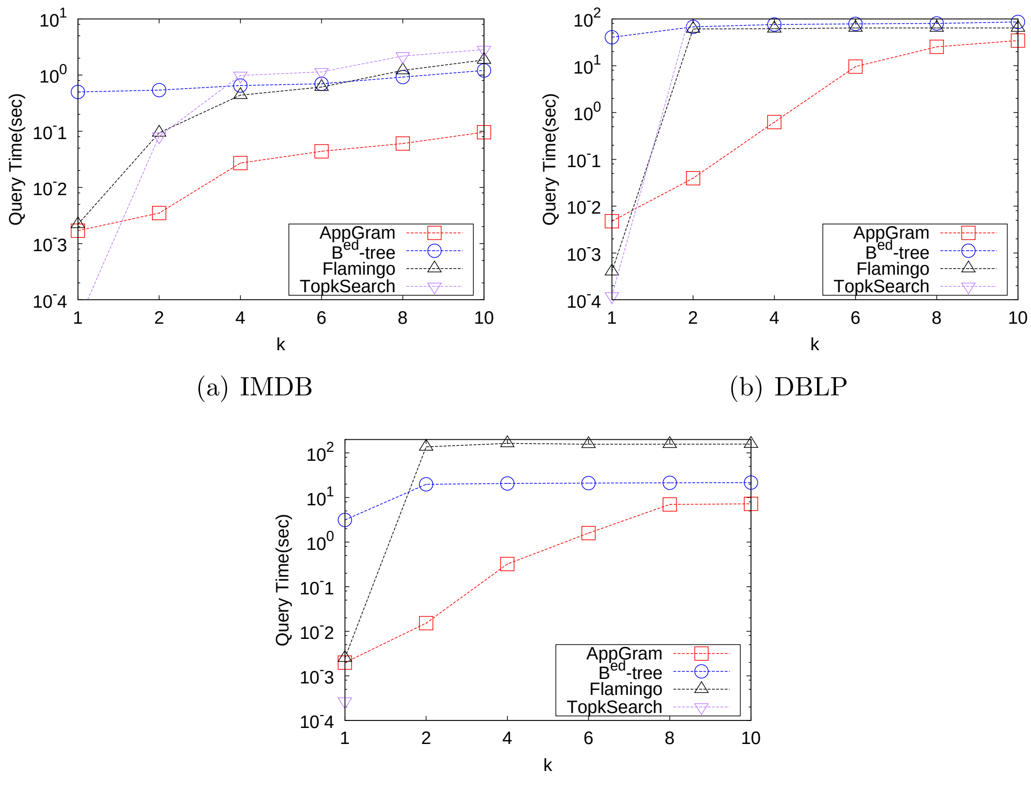


(a) IMDB (b) DBLP

(c) ANNOTEXT

Figure 4.12: Average candidate size vs. k

Figure 4.12 shows the candidate size for each filter combination. The results indicate that combining the MergeSkip with the length filter can help to reduce the candidate size and improve the query performance. Our proposed filter can further reduce the candidate size. Obviously, the proposed filtering technique can help to enhance the query performance of the MergeSkip technique and the length filter, as this filter combination need access to the smallest number of entries in the invert lists and generates the smallest number of candidates.



(a) IMDB (b) DBLP (c) ANNOTEXT

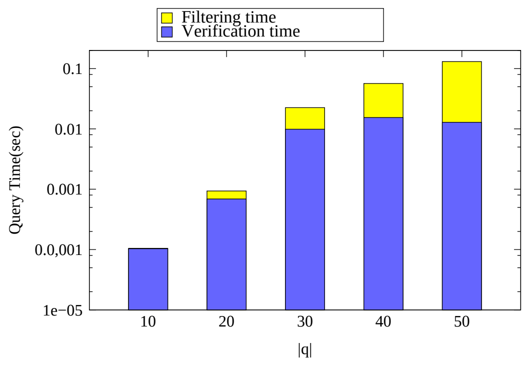
Figure 4.13: Average query time vs. k

4.6.5 Query Evaluation

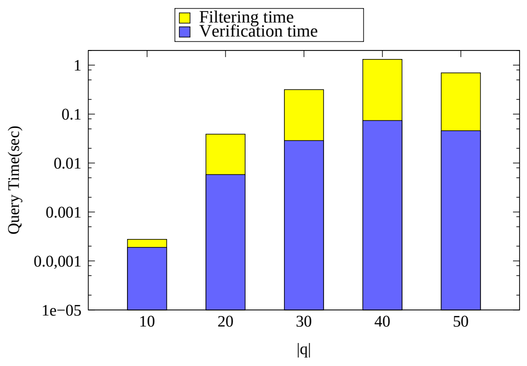
We evaluate the query performance of the proposed approach compared to exiting methods of B^{ed}-tree, Flamingo, and TopkSearch. The KNN search are conducted as follows. We vary k over 1, 2, 4, 8, and 16. For each k, we execute the 100 queries, and compute the average of the query results.

Figure 4.13 shows the average query time with respect to k on three datasets. It can be seen that the AppGram method outperforms all the competitive techniques with $k \geq 2$. The proposed f-queue and CA based filtering strategies can reduce the cost of the candidate verification. When the k value is as small as 1, Flamingo can run efficiently as it only needs to execute a range query once to obtain the top-1 result. In this case, a small edit distance threshold of 1 may be enough to report the top-1 sequence. Similarly, the TopkSearch is also more efficient for k=1 as its range-based algorithm only needs to verify a small number of entries in the dynamic matrix. However, when the k value increases, our AppGram method indeed outperforms other algorithms, and has a stable average query time. Note that TopkSearch takes too long to run on the long sequences of DBLP and the ANNOTEXT datasets and its results are excluded.

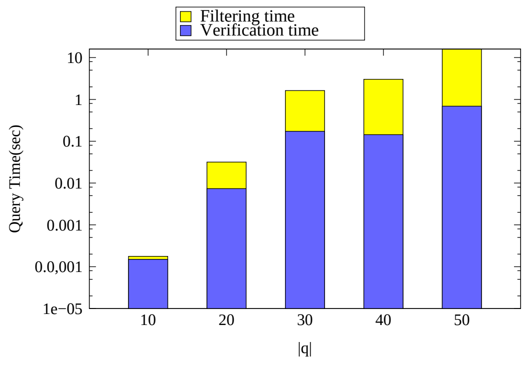
Figure 4.14 compares the overhead of two query phases: the filtering time and the candidate verification time. We randomly select five groups of queries with various average lengths of 10, 20, 30, 40, and 50, and each group has 100 query sequences. For each group, we execute the KNN search, and compute the average filtering time and verification time. It is clear that candidate verification is the most consuming step in the KNN search. That means that a tighter bound is required for speeding up KNN search and it is reasonable to sacrifice slightly higher filtering overhead to reduce the candidate sequences as much as possible.



(a) IMDB



(b) DBLP



(c) ANNOTEXT

Figure 4.14: Detailed analysis on the query cost vs. $|q|$

4.7 Summary

In this paper, we study the problem of k-nearest neighbor sequence search based on the edit distance. After surveying existing work, we find that such approaches have limitations on the applications, and often suffer poor filtering power and low query performance when sequences in the database are long. To speed up the KNN sequence search, we propose a novel pipeline approach using approximate n-grams. The approach follows a filter-and-refine framework. In the filtering phase, we develop a novel filtering technique based on counting the number of approximate n-grams, and experimental results show that this technique has high quality for pruning unqualified candidates. We also design an efficient searching algorithm with the frequency queue and the CA strategy. The frequency queue supports our proposed filtering techniques by reducing the number of candidate verification. By using the summation of gram edit distances as the aggregation function, the CA based search has an optimal feature of early termination which helps to invoke the halting condition of the whole pipeline framework. Our proposed filtering strategies have significant performance on the KNN search, and the pipeline framework is easy to support parallelism strategies.

CHAPTER 5

Readpeer: A Collaborative Annotation Cloud Service for Social Reading

Our project on social reading systems requires a unified indexing and query processing system to manage data resources and support real tasks, including annotation cloud services, book detection and recommendation, and so on. In general, data resources collected from such systems are often modeled as complex structures. We have successfully designed inverted list based index for supporting similarity queries on complex structure data like graphs and sequences. Based on the graph model and sequence model, we propose a novel data structure, denoted by nested structure, to model complex objects like ebook documents for handling some real tasks. We also design a unified inverted index for managing such complex and nested structures. The proposed 3-in-1 indexing system can provide a generic interface for inverted list storage, and help to support various types of queries on complex structures.

5.1 Overview

Over the last year, we have designed, implemented, and deployed a social reading system, denoted by **Readpeer**, for supporting users' social reading activities, such as online bookmarking, passage highlighting, comment sharing, and annotation retrieval etc. We aim to provide a user-friendly information management and sharing tool by collecting and organizing valuable resources such as highlights, annotations, sticky notes, documents, images, and videos, etc. Figure 5.1 presents such a recent reading system including basic reading features and social features. However, limited progress has been made because the management of such data resources is challenging. Such data cannot be simply organized using the traditional data management system such as relational databases. It poses new challenging research problems that do not exist in traditional databases. For example, how to organize these data? How to make them searchable for future retrieval? How to excite users with interesting knowledge discovery?

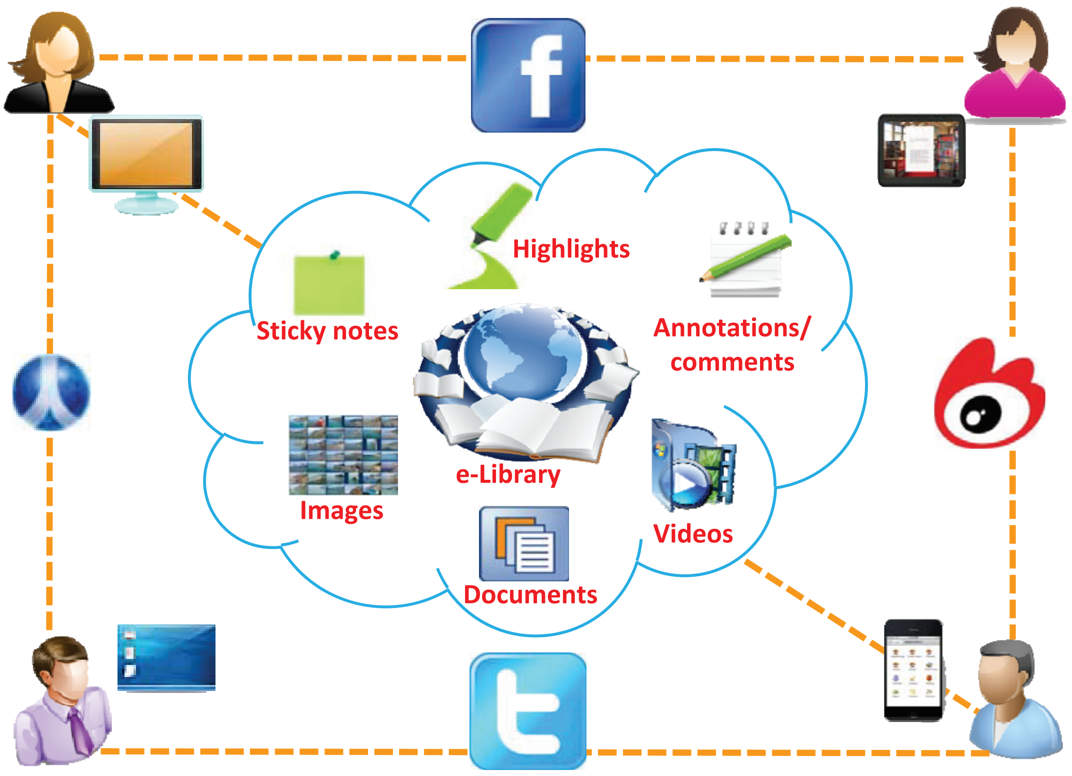


Figure 5.1: A recent social reading system

We demonstrate a working system that addresses these issues by providing the collaborative annotation cloud service. To manage the data resources, we use the complex structures to model various types of data. For example, we use the graph model to represent the documents, and the sequence model to represent the highlights and annotations. Based on these representations, the book detection problem could be formulated as the graph matching problem; while the annotation retrieval problem could be defined as the sequence similarity search problem. Our previous works described in Chapter 3 and 4 have made important progress to handle such problems. Our approach on graph similarity search can support efficient document duplicate detection, especially for those users with common interests who prefer to upload the same book. However, sometimes this approach fails to detect those books with multiple editions. In this case, different editions may have highly different graph representations. To solve such problems, we use a novel data structure, denoted by nested structure, to model a book document. See the example provided in Figure 1.6 in Section 1.1.4. A document can be modeled as a nested graph, whose vertices are sequences of titles, author names, and headings etc. With the model, we also define a similarity measure between two nested structures and design an index mechanism to handle the above task.

Even when we have developed efficient algorithms to process various types of complex and nested structures, it is still a waste of resources to design isolated indexing and processing systems for each kind of data. In this work, we propose a unified indexing system to support the storage and retrieval of all types of data. The proposed index mechanism is based on inverted lists. The basic idea is that we decompose original complex data into various types of sub-units, and use inverted files to store them. For example, for sequence data, we employ the n-gram decomposition and indexing method proposed in our previous work in Chapter 4. While for graph data, we use the star based approach in our previous work in Chapter 3. In general, a nested structure could be seen as a two-layer graph model including vertices of smaller complex structures like subgraphs and sequences. Therefore, a nested structure can be first decomposed into substructures of its vertices. Then, each vertex which may be a subgraph or a sequence, can be further decomposed into sub-units like stars or n-grams. Based on the decomposition, we use a two-level inverted index to store all types of sub-units. The proposed unified index structure can support various queries for different types of complex and nested structures. We simply implement the proposed indexing mechanism and use it to technically support the real tasks in our Readpeer system.

With the technical support, our social reading system brings valuable enticing features over existing systems, including 1) elegant storage of the data resources; 2) powerful annotation retrieval from anywhere; 3) interesting book and annotation discovery and expertise recommendation; 4) user collaboration with common interests. Our system provides a dream information manage tool for both personal reading networks and public social reading systems.

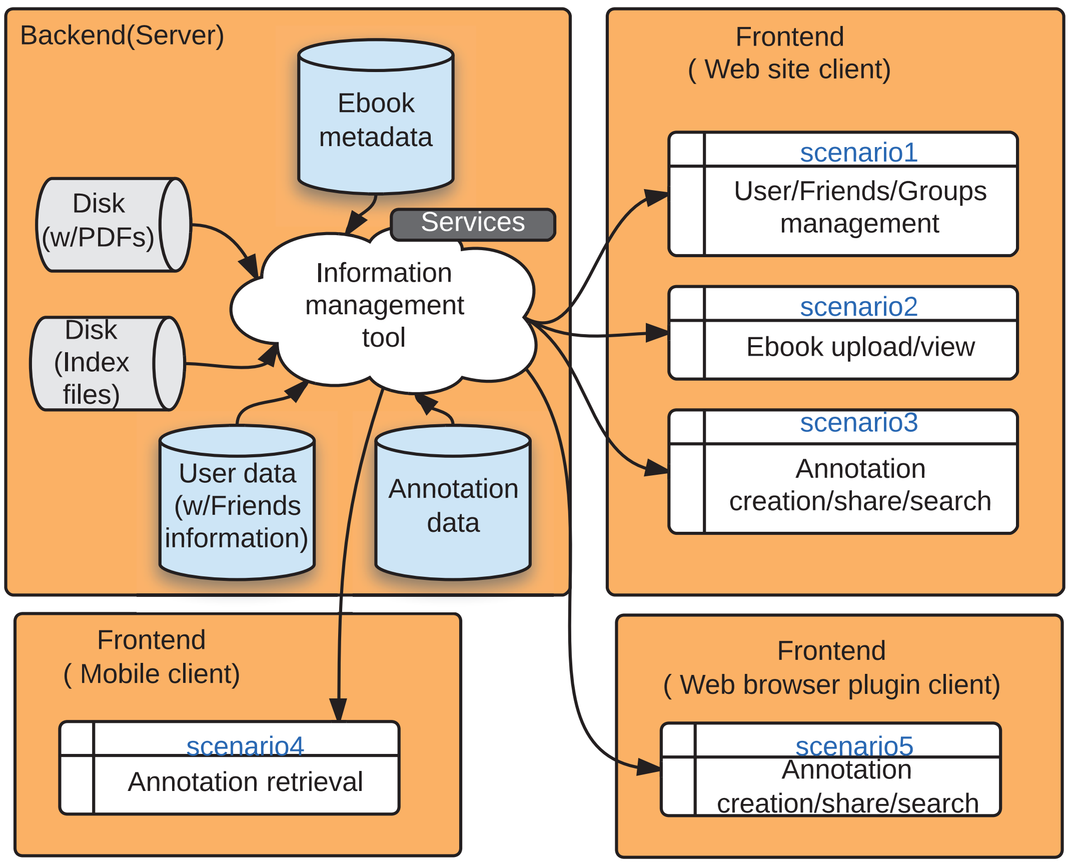

Backend(Server)
Ebook metadata
Disk (w/PDFs)
Disk (Index files)
Services
Information management tool
User data (w/Friends information)
Annotation data
Frontend (Web site client)
scenario1
User/Friends/Groups management
scenario2
Ebook upload/view
scenario3
Annotation creation/share/search
Frontend (Mobile client)
scenario4
Annotation retrieval
Frontend (Web browser plugin client)
scenario5
Annotation creation/share/search

Figure 5.2: System architecture

5.2 System Design

Figure 5.2 illustrates the architecture of our Readpeer system. The Readpeer system employs a client-server architecture. We build a user-friendly information management tool at the server-side, that supports the storage and retrieval of the user data. While at the client-side, we set up a web site, create an iOS app, and develop web browser plugins, for users to add, share, view, and retrieve annotations.

At the server-side, the collected user data are kept in a relational database management system (RDBMS), MySQL Server 5.1. Such user data include the information of users, the metadata of ebooks, and the annotation data etc. In addition, the original ebook files are stored in the file system. To provide flexible and intuitive access to the data, we build an information management tool between the relational database and the application layer. The tool employs the associated techniques to model, store, compare, and query the user data. The aim is to develop a unified indexing and query processing system to support efficient annotation retrieval and ebook detection. As shown in Figure 5.3, we implement the tool with three layers: the storage layer, the index layer, and the application layer. In the storage layer, we use complex structures to model the user data and store the complex structure databases in the file system. In the index layer, we use inverted lists to index various types of complex structures, based on the sub-unit decomposition method. We also design unified inverted list processing method to support query processing for various complex structures. The application layer is the processing layer where each type of complex structure can build specific processor to communicate with the other two layers.

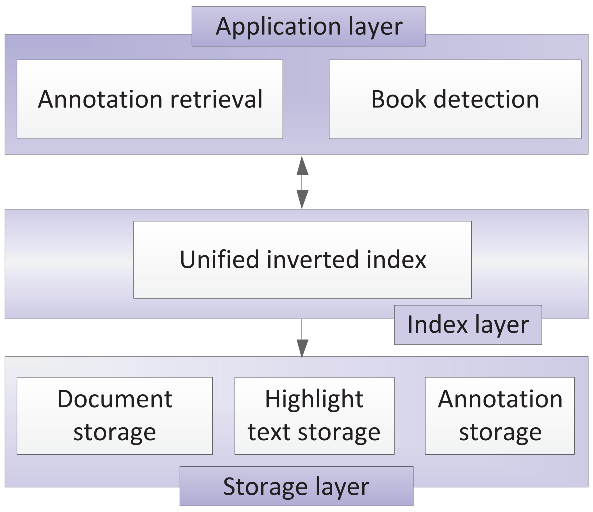


Figure 5.3: Information management tool architecture

5.2.1 Data Model

The implemented information management tool provides an appropriate storage for the data collected from Readpeer. The data include passages, comment texts, and documents etc. Various data structures are required to model various types of the data.

Passages and Comment Texts

Passages are collected from users' highlighting texts. They are often attached with several comments, notes, or media. Such passages and comment texts can be simply modeled as sequences. As mentioned in previous Section 1.1.2, the real application of such data models is the annotation search by snapping.

Documents

In general, documents are modeled as graphs. Figure 5.4 shows an example of such a document graph. Typically, a document might contain a title, author names, an abstract, and section headings. We build a simple document graph with vertices of title and headings, and add edge from a preceding heading to a succeeding heading. In this model, title and headings are hashed into numeric labels of vertices. Therefore, the ebook duplicate detection task can be supported using a graph range query algorithm. However, this model can not handle the ebook edition detection task, as different editions may have different graph representations. All the document elements including title and headings may have been updated into a new version. We motivationally propose a novel data model, denoted by nested structure, to model the document editions. A nested structure is defined as a graph with vertices of sequences. See the example in Figure 5.4. The difference between the nested structure and the graph is that, the title and headings in the nested graph are represented as sequences instead of hash codes in the simple graph.

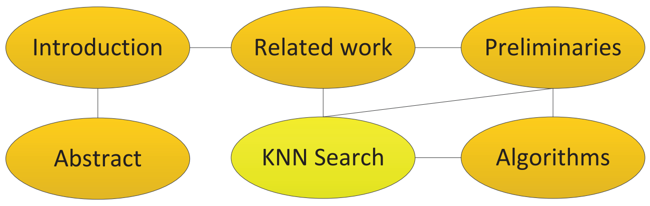


Figure 5.4: A document graph

5.2.2 Unified Inverted Index

We have used sequence, graph, and nested structure to model the collected data like annotation texts and documents. Based on our previous works in Chapter 3 and 4, we know that the inverted list based index can support efficient query processing for both graph model and sequence model. As shown in Figure 5.5, a unified 3-in-1 inverted index can be built by breaking non-nested complex structures down into smaller units like stars (for graphs [67]), n-grams (for sequences [68]), and binary branches (for trees [?]). With this index, searches are performed by retrieving these smaller units individually and assembling them for matching the queries.

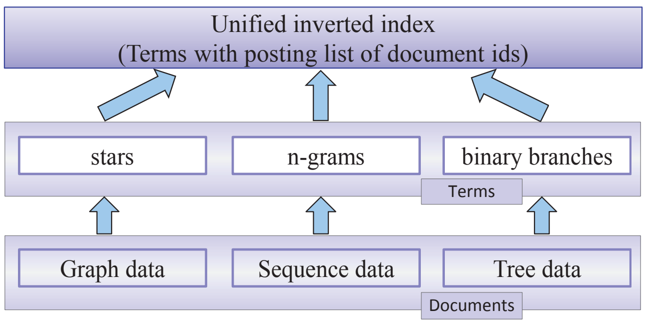


Figure 5.5: A unified inverted index structure

We investigate how to extend the above inverted index to support query processing on nested structures. The basic idea is adapted from our previous works on graph and sequence model. We build a two layer inverted index: 1) In the outer layer, the nested document graph is first broken into a multi-set of stars using the star decomposition method proposed in our previous work in Chapter 3. Here, each vertex of the document graph is labelled using both the hash code and the original sequence. The hash code is used as the labels in the star structure, and an inverted index is also built to store the reference between graphs and stars. 2) In the inner layer, for each vertex of the document graph, we break its sequence into a multi-set of n-grams. An inverted index is built to store the reference between vertex sequences and n-grams.

Obviously, the inverted index can be used to store all types of data including sequences, trees, graphs, and nested structures. The need arises only to support the storage of inverted files for efficient query processing. In this work, we implement the unified storage on top of the Apache Lucene[1]. In our implementation, the index structure is organized as an inverted manner from terms to the list of documents (which contain the term). The list (known as posting list) is ordered by a global ordering (typically by document id). Each inverted index is stored on disk as segment files which will be brought to memory during the list processing.

[1]http://lucene.apache.org

5.2.3 Data Queries

We illustrate how to use the inverted index to support the similarity queries on various complex structures. In this work, we implement a unified CA-based list processing algorithm for our applications. The query is submitted from the application layer with a configuration file to the index layer. The configuration file includes all the required information for processing a specific query, such as the location of index file, the location of data file, the similarity measure, the similarity threshold value, the query type, and the aggregation function. The index layer will load the corresponding index file into the memory with the location provided in the configuration file. Then, list processing interface is called to answer the query.

We illustrate how to evaluate the similarity between two nested structures and use the proposed index structure to support the efficient query processing on nested structures. In this work, we focus on the nested structures on document graphs. We use graph edit distance to evaluate the similarity between two document graphs. Given a query nested structure, we generate the hash codes for each vertex, and decompose the query into a multi-set of stars. For each vertex sequence, we decompose it into a multi-set of n-grams and perform a KNN sequence search on the inverted index in the inner layer. The top-k results for each vertex will be returned to the outer layer. Then, the top-k sequences will be considered as the matching vertices, especially when computing the star edit distance. Finally, graph similarity search algorithms are employed to support the final candidate verification.

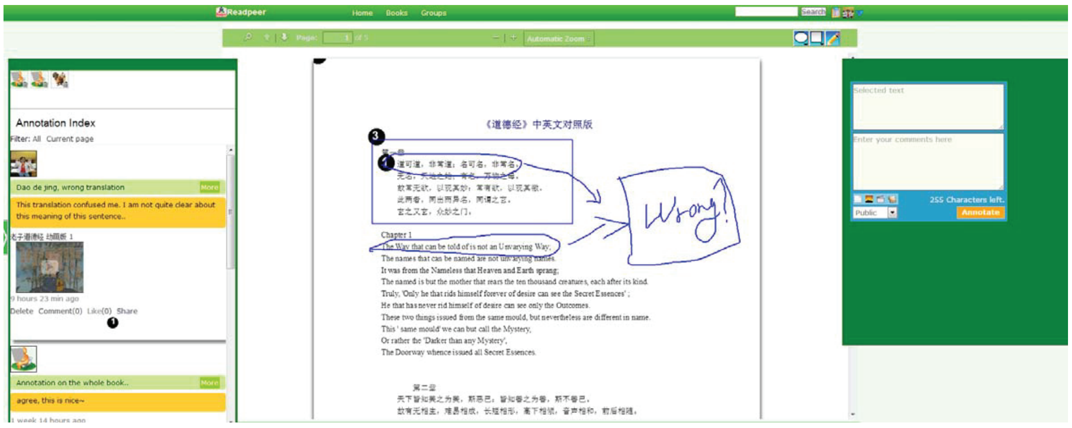
In summary, we implement the unified indexing and query processing system into the information management tool, which can efficiently support real tasks in the Readpeer system.

5.3 System Demonstration

At the client-side, we have provided three types of accesses to use our Readpeer system: web site, iOS app, and web browser plugin.

5.3.1 Readpeer Web Site

The Readpeer web site is now launched in the first public version of Readpeer.com[2]. It consists of important reader features, including ebook uploading, passage highlighting, comment sharing, and annotation searching etc. It also implements a number of social features, such as sharing comments with friends, activating connections between users, presenting activity feeds, and creating reading groups etc.



Dao de jing, wrong translation

This translation confused me. I am not quite clear about this meaning of this sentence..

老子道德经 动画版 1

9 hours 23 min ago

Annotation on the whole book..

agree, this is nice~

1 week 14 hours ago

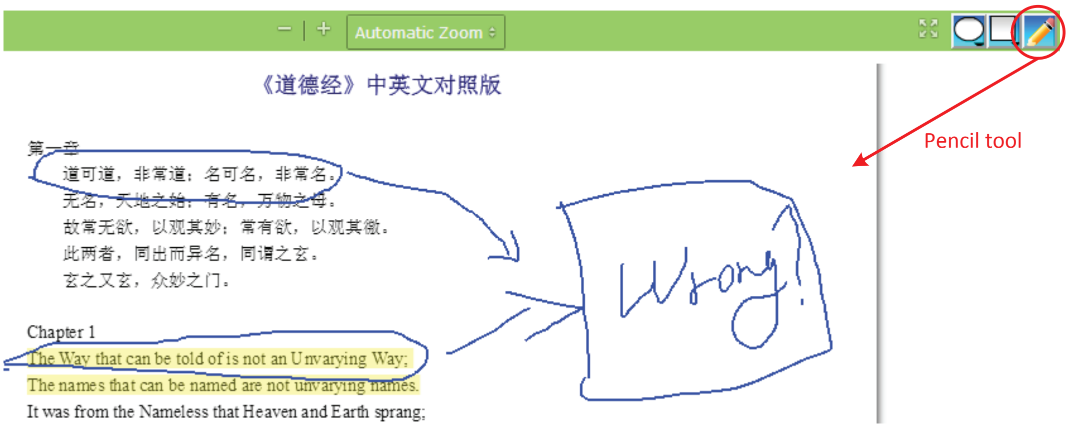
《道德经》中英文对照版

第一章
道可道，非常道；名可名，非常名。
无名，天地之始；有名，万物之母。
故常无欲，以观其妙；常有欲，以观其徼。
此两者，同出而异名，同谓之玄。
玄之又玄，众妙之门。

Chapter 1
The Way that can be told of is not an Unvarying Way;
The names that can be named are not unvarying names.
It was from the Nameless that Heaven and Earth sprang;
The named is but the mother that rears the ten thousand creatures, each after its kind.
Truly, 'Only he that rids himself forever of desire can see the Secret Essences';
He that has never rid himself of desire can see only the Outcomes.
These two things issued from the same mould, but nevertheless are different in name.
This 'same mould' we can but call the Mystery,
Or rather the 'Darker than any Mystery',
The Doorway whence issued all Secret Essences.

第二章
天下皆知美之为美，斯恶已；皆知善之为善，斯不善已。
故有无相生，难易相成，长短相形，高下相倾，音声相和，前后相随。

Figure 5.6: Current ebook reader



《道德经》中英文对照版

第一章
道可道，非常道；名可名，非常名。
无名，天地之始；有名，万物之母。
故常无欲，以观其妙；常有欲，以观其徼。
此两者，同出而异名，同谓之玄。
玄之又玄，众妙之门。

Chapter 1
The Way that can be told of is not an Unvarying Way;
The names that can be named are not unvarying names.
It was from the Nameless that Heaven and Earth sprang;

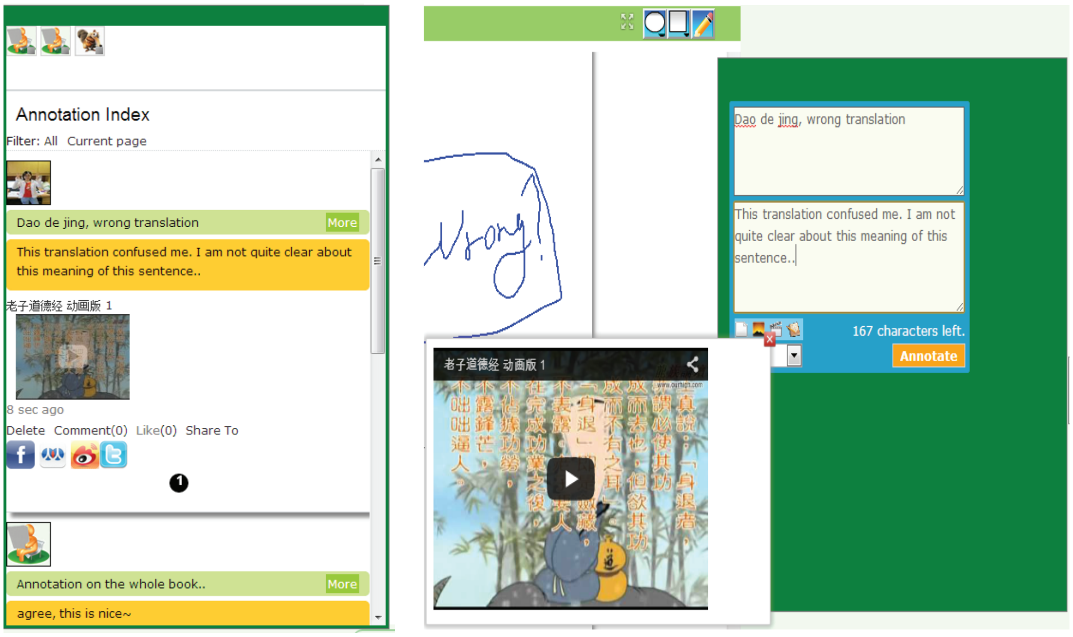
Figure 5.7: Highlights with the pencil tool

Figure 5.6 presents an overview of the ebook reading interface. The current design has three columns. To be a central, dominant component, the ebook is rendering in the center column. We also present previous highlighted passages

[2]http://Readpeer.com

in the current page. In the left column, we add a left sidebar to show user's friends who has highlighted something and share comments in the current page. In addition, all previous annotations and comments are also presented here. In the right column, the annotation box can automatically show up in the right siding bar when the user has highlighted something in the current page. To provide a more user-friendly interface, both the two siding bars can be opened and closed, to hide and show corresponding contents.

The Readpeer system provides three types of canvas annotation tools: rectangle tool, ellipse tool and pencil tool. With these tools, users are convenient to highlight passages and add annotations everywhere in the book. Figure 5.7 shows an example of using the pencil tool.



Dao de jing, wrong translation

This translation confused me. I am not quite clear about this meaning of this sentence..

Annotation on the whole book..

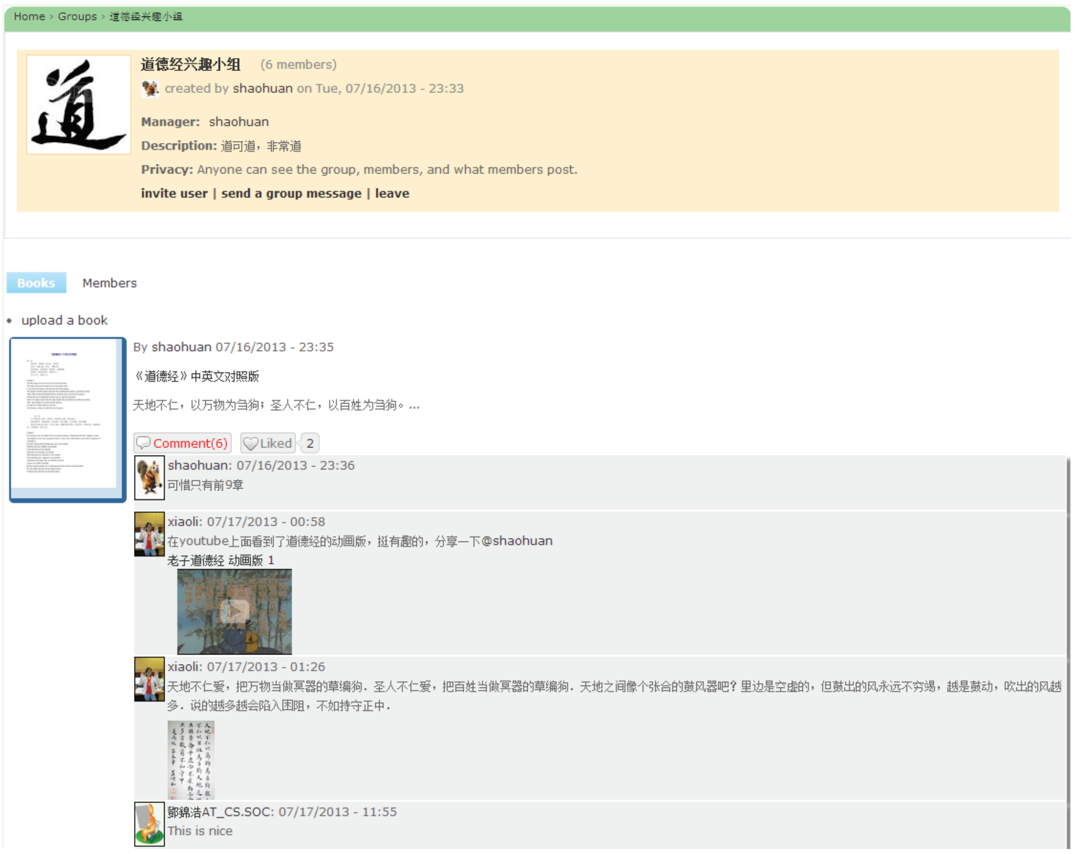
agree, this is nice~

(a) Read a comment (b) Write a comment

Figure 5.8: Current comment interface

When the user highlights something, the annotation box will open in the right siding bar. The user can type in a note to attach comments to the highlight. Comments may be public to all users, friends, groups, or private. Some readers make private comments like sticky notes, while others use the public comments to ask questions. The user also can attach associated web pages, media, and blogs into the comments. Figure 5.8 (b) shows an example to attach a video into a comment. When reading comments from others, the user can click the comment button to reply others' comments. Figure 5.8 (a) lists the comments in the current page. The use can like the annotations, and share the interesting annotations to public social web sites like Facebook[3], Twitter[4], and Sina Weibo[5].

Beside the ebook reader, we also build a group reading system. Figure 5.9 presents a reading group created by a user in our site. In our system, users can create many reading groups, invite friends or other users to join the group, add new books to the reading list, and share comments within a book. The group system provides a convenient connection way for users to find people with common interests and share their ideas.



Home › Groups › 道德经兴趣小组

道德经兴趣小组 (6 members)
created by shaohuan on Tue, 07/16/2013 - 23:33

Manager: shaohuan
Description: 道可道，非常道
Privacy: Anyone can see the group, members, and what members post.
invite user | send a group message | leave

Books Members

- upload a book

By shaohuan 07/16/2013 - 23:35
《道德经》中英文对照版
天地不仁，以万物为刍狗；圣人不仁，以百姓为刍狗。...

Comment(6) Liked 2

shaohuan: 07/16/2013 - 23:36
可惜只有前9章

xiaoli: 07/17/2013 - 00:58
在youtube上面看到了道德经的动画版，挺有趣的，分享一下@shaohuan
老子道德经 动画版 1

xiaoli: 07/17/2013 - 01:26
天地不仁爱，把万物当做冥器的草编狗．圣人不仁爱，把百姓当做冥器的草编狗．天地之间像个张合的鼓风器吧？里边是空虚的，但鼓出的风永远不穷竭，越是鼓动，吹出的风越多．说的越多越会陷入困阻，不如持守正中．

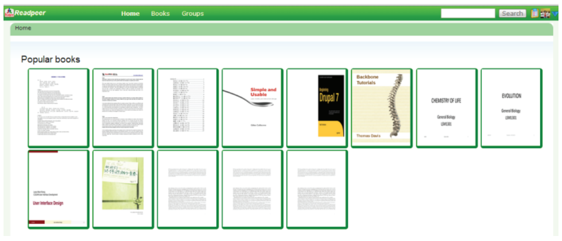
鄧錦浩AT_CS.SOC: 07/17/2013 - 11:55
This is nice

Figure 5.9: An example of reading group

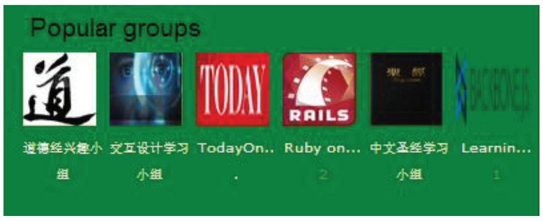
Readpeer also provides a recommendation system based on knowledge discovery on interesting book and annotation discovery. Four types of recommendations, including the popular books, groups, expertise, and annotations, are

[3]https://www.facebook.com
[4]https://twitter.com
[5]http://www.weibo.com

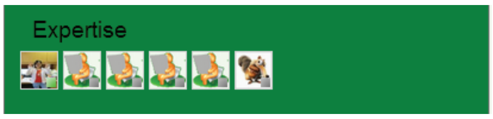
separately presented from Figure 5.10 (a) to Figure 5.10 (d). For books and annotations, the ranking function is computed as $rank = \alpha \times c_l + \beta \times c_a$. c_l and c_a denote the number of likes and annotations. α and β denote the assigned weights for them. We compute a score for a user with the total number of posts and active conditions. For groups, the ranking function is computed as $rank = \alpha \times \overline{r_m} c_m + \beta \times \overline{r_b} \times c_b$. Here, c_m and c_b denote the number of members and books in this group. $\overline{r_m}$ and $\overline{r_b}$ are the average ranking scores of members and books. α and β are the assigned weights.



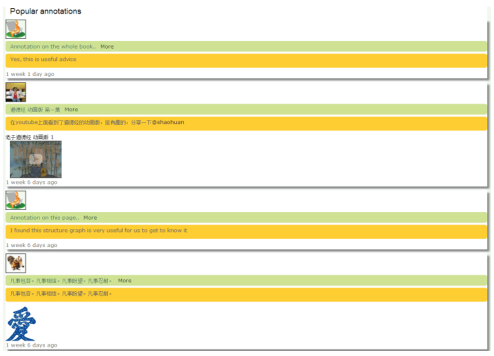
(a) Popular books



(b) Popular groups



(c) Popular expertise



(d) Popular annotations

Figure 5.10: Popular blocks

Content on Readpeer comes from various sources, including free public web pages, ebooks, and slides etc. Much activity on Readpeer currently center on slides from teachers and students in School of Computing in National University of Singapore. Teachers and students upload their slides to our site, and add comments or notes to their slides. Readpeer helps students to better capture the gist of the teacher's lecture, and provides teachers a better management tool for managing their courses. It also can be used as an online component of the classrooms in the university. Another important source for Readpeer is the ebook library from Netease Cloud Reading[6]. They have provided a complete elibrary of numerous books, newspapers, magazines, and web pages etc.

[6]http://yuedu.163.com/?act=rdwzb_20121221_01

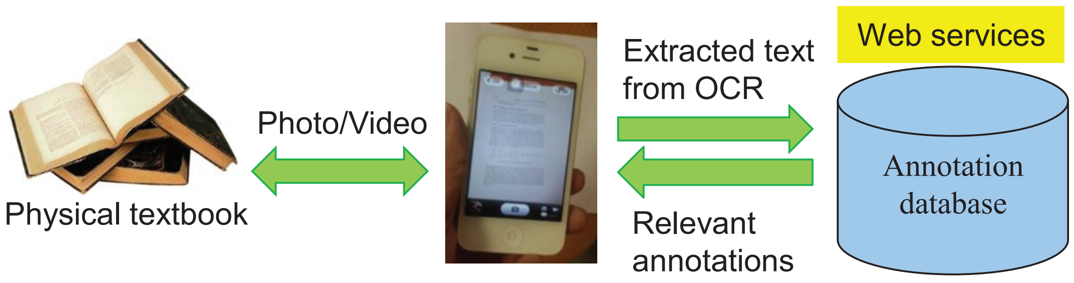
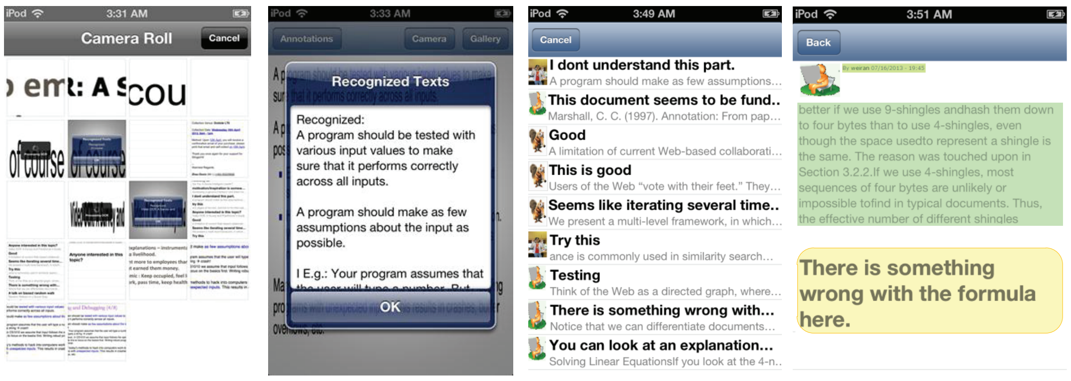


Figure 5.11: An example of annotation retrieval



Recognized:
A program should be tested with various input values to make sure that it performs correctly across all inputs.

A program should make as few assumptions about the input as possible.

I E.g.: Your program assumes that

I dont understand this part.
A program should make as few assumptions…
This document seems to be fund..
Marshall, C. C. (1997). Annotation: From pap…
Good
A limitation of current Web-based collaborati…
This is good
Users of the Web “vote with their feet.” They…
Seems like iterating several time..
We present a multi-level framework, in which…
Try this
ance is commonly used in similarity search…
Testing
Think of the Web as a directed graph, where…
There is something wrong with…
Notice that we can differentiate documents…
You can look at an explanation…
Solving Linear EquationsIf you look at the 4-n..

By weiran 07/16/2013 - 19:45

better if we use 9-shingles andhash them down to four bytes than to use 4-shingles, even though the space usedto represent a shingle is the same. The reason was touched upon in Section 3.2.2.If we use 4-shingles, most sequences of four bytes are unlikely or impossible tofind in typical documents. Thus, the effective number of different shingles

There is something wrong with the formula here.

(a) Select a snapshot (b) Extract text from OCR (c) Search the annotations (d) Read an annotation

Figure 5.12: Screen captures on the iOS app

5.3.2 The iOS App

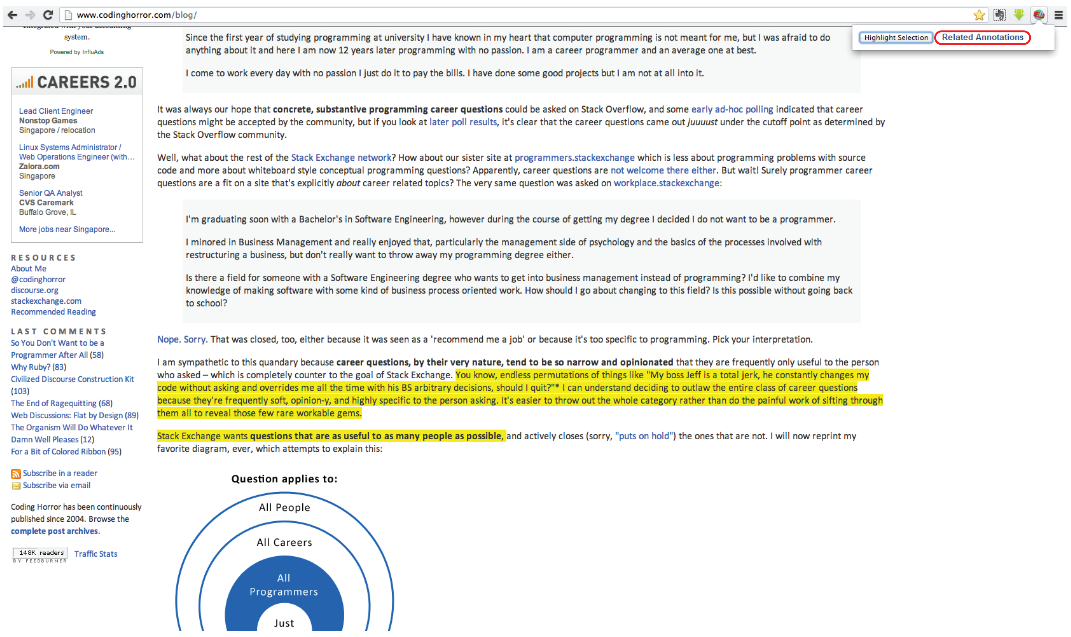
We create an iOS frontend app to support efficient annotation retrieval and recommendation. Our app provides an interesting feature to detect annotations on physical books through augmented reality. Figure 5.11 shows an example of the use of OCR to search for annotations. As mentioned before, an information management tool is developed in the server side to provide web services for annotation search. As shown in Figure 5.12, users can use the iOS app with four steps as below.

- **Select a snapshot:** Users use mobile devices to snap a photo of page in real books and pick a snapshot as a query photo, as shown in Figure 5.12 (a).

- **Extract text from OCR:** The query photo is then processed by an optical character recognition (OCR) program which extracts the text from the photo as a sequence, as shown in Figure 5.12 (b).

- **Search the annotations:** The iOS app sends a request to the web services to search associated annotations with the query sequence. Note that the OCR program might generate errors within the sequence. The information management tool needs to perform an approximate query against the passages in the server to retrieve those passages that had been annotated. The returned results are shown in Figure 5.12 (c).

- **Read an annotation:** Users also can click on one returned annotation to view the details of annotated passage and related comments, as shown in Figure 5.12 (d).



system.

Powered by InfluAds

CAREERS 2.0

Lead Client Engineer
Nonstop Games
Singapore / relocation

Linux Systems Administrator / Web Operations Engineer (with…
Zalora.com
Singapore

Senior QA Analyst
CVS Caremark
Buffalo Grove, IL

More jobs near Singapore...

RESOURCES
About Me
@codinghorror
discourse.org
stackexchange.com
Recommended Reading

LAST COMMENTS
So You Don't Want to be a Programmer After All (58)
Why Ruby? (83)
Civilized Discourse Construction Kit (103)
The End of Ragequitting (68)
Web Discussions: Flat by Design (89)
The Organism Will Do Whatever It Damn Well Pleases (12)
For a Bit of Colored Ribbon (95)

Subscribe in a reader
Subscribe via email

Coding Horror has been continuously published since 2004. Browse the **complete post archives**.

148K readers BY FEEDBURNER Traffic Stats

> Since the first year of studying programming at university I have known in my heart that computer programming is not meant for me, but I was afraid to do anything about it and here I am now 12 years later programming with no passion. I am a career programmer and an average one at best.
>
> I come to work every day with no passion I just do it to pay the bills. I have done some good projects but I am not at all into it.

It was always our hope that **concrete, substantive programming career questions** could be asked on Stack Overflow, and some early ad-hoc polling indicated that career questions might be accepted by the community, but if you look at later poll results, it's clear that the career questions came out *juuuust* under the cutoff point as determined by the Stack Overflow community.

Well, what about the rest of the Stack Exchange network? How about our sister site at programmers.stackexchange which is less about programming problems with source code and more about whiteboard style conceptual programming questions? Apparently, career questions are not welcome there either. But wait! Surely programmer career questions are a fit on a site that's explicitly *about* career related topics? The very same question was asked on workplace.stackexchange:

> I'm graduating soon with a Bachelor's in Software Engineering, however during the course of getting my degree I decided I do not want to be a programmer.
>
> I minored in Business Management and really enjoyed that, particularly the management side of psychology and the basics of the processes involved with restructuring a business, but don't really want to throw away my programming degree either.
>
> Is there a field for someone with a Software Engineering degree who wants to get into business management instead of programming? I'd like to combine my knowledge of making software with some kind of business process oriented work. How should I go about changing to this field? Is this possible without going back to school?

Nope. Sorry. That was closed, too, either because it was seen as a 'recommend me a job' or because it's too specific to programming. Pick your interpretation.

I am sympathetic to this quandary because **career questions, by their very nature, tend to be so narrow and opinionated** that they are frequently only useful to the person who asked – which is completely counter to the goal of Stack Exchange. You know, endless permutations of things like "My boss Jeff is a total jerk, he constantly changes my code without asking and overrides me all the time with his BS arbitrary decisions, should I quit?"* I can understand deciding to outlaw the entire class of career questions because they're frequently soft, opinion-y, and highly specific to the person asking. It's easier to throw out the whole category rather than do the painful work of sifting through them all to reveal those few rare workable gems.

Stack Exchange wants **questions that are as useful to as many people as possible**, and actively closes (sorry, "puts on hold") the ones that are not. I will now reprint my favorite diagram, ever, which attempts to explain this:

Figure 5.13: Current web browser plugin

5.3.3 Web Browser Plugin

To build a scale information management tool for collaborative cloud services, it is important to provide an interface for collecting the user data as many as possible. A web browser plugin can achieve such goals with a simple implementation. Figure 5.13 shows the interface of our current web browser plugin. The plugin provides several useful functionalities for users: open annotation sharing, tracking, and searching. Anyone can view previous open annotations in any public web page. Users are authorized using the the OAuth 2.0 protocol with Readpeer accounts, or public social network accounts like Facebook or Twitter. When passing the authorization, the user can add public annotations and search associated annotations. As shown in Figure 5.13, the user can highlight text from any public web page, attach comments or notes into the highlight, and store the data with the page link into the Readpeer database. If other users visit the same web page, previous annotations associated with the link will be returned. Users can click the "Related Annotations" button to track these comments and notes. If no result is returned for current page link, a backend thread will be activated to send a request for searching associated annotations. In this case, all texts in current web page are extracted to build a long query sequence. The server side will process the query sequence, and perform a KNN search to retrieve those passages that had been annotated. The returned results will be automatically shown in an open side bar.

5.4 Summary

We presented our Readpeer system, which is a social reading system that supports a number of reader features and social features. The system follows a server-client model. In the client side, we have built three access platforms to use the Readpeer system, including the web site, the iOS app, and the web browser plugin. Readpeer connects users across such reading platforms. It provides the elegant interface to visualize users' highlights and annotations. In the server side, we have developed a powerful information management tool to support the annotation retrieval, ebook duplicate detection, and ebook edition detection. To support efficient annotation retrieval, we employ the KNN sequence search algorithm that is proposed in our previous work on sequence similarity search. To support ebook duplicate detection, we use the graph range query algorithm that is proposed in our previous work on graph similarity search. To support ebook edition detection, we use a new data model, denoted by nested structure, to model the documents. We also adapt the graph range query algorithm to support the nested structure search. For the annotation retrieval, the query performance has been evaluated in the ANNOTEXT dataset in our previous work. As Readpeer executes the ebook detection tasks as time-triggered actions, the query results will be finally verified by system administrators or project team members. Therefore, the efficiency of such tasks is not our focus on this work. To avoid the waste of resources, we also develop a unified 3-in-1 indexing system for supporting efficient storage and retrieval of various complex or nested structures. In summary, the 3-in-1 system provides an powerful information management tool for Readpeer.

CHAPTER 6

Conclusion and Future Work

This dissertation developed a unified 3-in-1 indexing system to support query processing of various complex or nested structures, such as sequences, trees, and graphs. Three works have been proposed to solve the similarity search problems respectively on graphs, sequences, and nested structures. In the following subsections, we conclude three works based on the experimental results, and then discuss the possible avenues that can be undertaken in the future.

6.1 Graph Similarity Search

To investigate an important problem of GED based graph similarity search, we proposed SEGOS, an efficient indexing and pipeline query processing framework based on sub-units. Experimental results on two real datasets showed that the proposed algorithm returned the smallest number of candidates, by even 100 times less than the existing works. Compared to **C-Tree** [29], it is clear that SEGOS dominates **C-Tree** w.r.t response time and candidate size. Although **κ-AT** [63] answers queries using simple filtering techniques, it uses a very loose bound which can result in very poor filtering power. Based on this result, it makes sense to sacrifice a few more milliseconds to prune as many candidates as possible, because the GED computation is extremely expensive.

Based on the experimental evaluation, the proposed approach can outperform the state-of-the-art works with the best query processing performance.

The developed index structure extends previous work by reducing the access to those very dissimilar graphs, and can improve previous work by speeding up the graph search.

6.2 Sequence Similarity Search

To support edit similarity queries in a database with long strings, we proposed a novel pipeline framework using approximate n-grams. To store the approximate n-grams decomposed from database strings, a multi-level inverted index was constructed. At search time, the proposed strategy has been proposed to follow a parallel processing way with multiple pipelines. Extensive experiments over three real datasets showed that the proposed framework had the smallest query time in about 10 milliseconds in the KNN search. It was found that the new pipeline framework can prune much more graphs than existing works. The query processing time was found to be significantly reduced using the parallel processing.

Nevertheless, the proposed framework works well on handling the KNN sequence search, which could be more important than the range query in practice. The advantage of the novel search strategy in this work over conventional ones is that the pruning power can be highly improved by relaxing the filtering condition and the CA search algorithm provides a better framework for the parallel query processing.

6.3 3-in-1 Indexing System

To investigate the properties of complex and nested structures based on graph model and sequence model, we developed a unified 3-in-1 index framework for various complex or nested structures. In the 3-in-1 system, a basic method to retrieve smaller sub-units to generate inverted lists was proposed such that all structures containing a particular sub-unit can be indexed by one of the inverted lists. Since there could be a large number of inverted lists, additional indexing mechanism has been developed to quickly access the relevant inverted list for a query. It was found that the proposed nested structure model can well represent complex objects in the real world. The results returned from queries were shown to be interesting, and have practical usages in real systems.

The developed indexing mechanism was found to be efficient and effective to support similarity search.

This is the first work to develop a unified 3-in-1 system that can be useful for supporting different complex structures. It also opens up new avenues that involve the model and search for a variety of complex and nested structures. This unified system has many applications in real life. We also present a real social reading system in http://readpeer.com to show how to employ the unified indexing and processing system to solve real problems in practice.

6.4 Future works

As the wide applications of complex structures, future works can be undertaken as follows.

The similarity search on trees is not our focus in our unified system. The reason is that we see a tree as a simple and specified graph. This makes sense because it is easy to extend the graph search algorithms for supporting the tree similarity search. Future works can be undertaken as how to extend the idea of our first work to handle the similarity search problems on trees.

The direct extension of our first work on graph search is to adapt the proposed bounds to theoretically support the sub-graph matching problems. In the literature, the sub-graph matching problem is very important for its wide applications in many research fields, like bio-informatics, chem-informatics, and so on. However, no efficient method has been proposed to solve this problem in all areas. It would be very useful if we can directly extend our work to efficiently handle this problem.

The new avenues on the model and search for nested structures have significant usages in practice. Since our third work is the first to address the problem, many interesting topics need to be further investigated. For example, queries like "Is there a document containing images or figures showing characteristics of cars?" cannot be simply represented as a traditional search pattern. How to model this kind of queries remains an open problem. This dissertation selects edit distance as the similarity measure between two complex structures. However, edit distance computation on nested structures can be very complex and costly. As the wide usage of edit distance, it is the first choice to extend it for measuring the similarity between two nested structures. While considering the complexity of edit distance, it would be more useful to define a more practical similarity measure. Existing works have proposed simple similarity measure like jaccard or cosine similarity. We also have a previous work on document join problems with various similarity measures [51], it is interesting to further investigate such issues on the document nested graphs.

In current Readpeer system, we implement the storage of the unified inverted index by adapting the Apache Lucene index. However, to provide a more powerful information management tool, we need to consider some important issues in inverted index, including index compression, incremental updates, and distributed query performance. There are many other database management systems that have provided such support for storing the inverted index, such as PostgreSQL[1] and ElasticSearch[2] etc. It is worthwhile to investigate such systems to support more effective storage of the inverted index. In particular, how to implement our storage system on top of distributed system is an interesting issue for improving the query performance.

Furthermore, our real social reading system is an ongoing and open project. There are still many challenging and theoretical problems existing in the annotation searching by snapping and the fast online text searching. Especially, as more and more annotation data are collected, powerful storage and query processing techniques are required. It is interesting and useful to further investigate all these practical problems. Moreover, our current information management tool cannot support processing the image and video data. However, such data have wide usage on real social media systems. Therefore, a more interesting and useful issue is to address the image and video processing problems using the proposed inverted index system.

[1]http://www.postgresql.org
[2]http://www.elasticsearch.org

Bibliography

[1] H. A. Almohamad and S. O. Duffuaa. A linear programming approach for the weighted graph matching problem. *IEEE Trans. Pattern Anal. Mach. Intell.*, 15(5):522–525, 1993.

[2] Sattam Alsubaiee, Alexander Behm, and Chen Li. Supporting location-based approximate-keyword queries. In *ACM GIS*, pages 61–70, 2010.

[3] Arvind Arasu, Venkatesh Ganti, and Raghav Kaushik. Efficient exact set-similarity joins. In *VLDB*, pages 918–929, 2006.

[4] Alexander Behm, Chen Li, and Michael J. Carey. Answering approximate string queries on large data sets using external memory. In *ICDE*, pages 888–899, 2011.

[5] P. Bille. A survey on tree edit distance and related problems. *Theor. Comput. Sci.*, 337(1-3):217–239, 2005.

[6] Eric W. Brown, James P. Callan, W. B Croft, and J. E.B. Moss. Supporting full-text information retrieval with a persistent object store. In *EDBT*, pages 365–378, 1994.

[7] Horst Bunke and Kim Shearer. A graph distance metric based on the maximal common subgraph. *Pattern Recognition Letter*, 19(3-4):255–259, 1998.

[8] X. Cao, S. C. Li, and A. K. H. Tung. Indexing dna sequences using q-grams. In *DASFAA*, pages 4–16, 2005.

[9] Surajit Chaudhuri and Raghav Kaushik. Extending autocompletion to tolerate errors. In *SIGMOD*, pages 707–718, 2009.

[10] James Cheng, Yiping Ke, Wilfred Ng, and An Lu. Fg-index: towards verification-free query processing on graph databases. In *SIGMOD*, pages 857–872, 2007.

[11] D. Cutting and J. Pedersen. Optimization for dynamic inverted index maintenance. In *ACM SIGIR*, pages 405–411. ACM, 1989.

[12] D. Cutting and J. Pedersen. Optimization for dynamic inverted index maintenance. In *ACM SIGIR*, pages 405–411, 1990.

[13] Samuel Defazio, Amjad M. Daoud, Lisa Ann Smith, and Jagannathan Srinivasan. Integrating ir and rdbms using cooperative indexing. In *ACM SIGIR*, pages 84–92, 1995.

[14] Deng Dong, Li Guoliang, Feng Jianhua, and Li Wen-Syan. Top-k string similarity search with edit-distance constraints. In *ICDE*, 2013.

[15] Ronald Fagin, Amnon Lotem, and Moni Naor. Optimal aggregation algorithms for middleware. In *PODS*, pages 102–113, 2001.

[16] Christos Faloutsos and H. V. Jagadish. On b-tree indices for skewed distributions. In *VLDB*, pages 363–374, 1992.

[17] Mirtha-Lina Fernńdez and Gabriel Valiente. A graph distance metric combining maximum common subgraph and minimum common supergraph. *Pattern Recognition Letter*, 22(6-7):753–758, 2001.

[18] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319–349, 1987.

[19] Mark Gabel, Lingxiao Jiang, and Zhendong Su. Scalable detection of semantic clones. In *ICSE*, pages 321–330, 2008.

[20] X. Gao, B. Xiao, D. Tao, and X. Li. A survey of graph edit distance. *Pattern Anl. & Applic.*, 2009.

[21] M.R. Garey and D.S. Johnson. *Computers and intractability.* Freeman San Francisco, 1979.

[22] Rosalba Giugno and Dennis Shasha. Graphgrep: A fast and universal method for querying graphs. In *ICPR*, pages 112–115, 2002.

[23] D. A. Gorssman and J. R. Driscoll. Structuring text within a relation system. In *DEXA*, pages 72–77, 1992.

[24] 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. Citeseer, 2001.

[25] Luis Gravano, Panagiotis G. Ipeirotis, H. V. Jagadish, Nick Koudas, S. Muthukrishnan, and Divesh Srivastava. Approximate string joins in a database (almost) for free. In *VLDB*, pages 491–500, 2001.

[26] S. Guha, H. V. Jagadish, N. Koudas, D. Srivastava, and T. Yu. Approximate xml joins. In *SIGMOD*, pages 287–298, 2002.

[27] J. Han, X. Yan, and P.S. Yu. Mining, indexing, and similarity search in graphs and complex structures. In *ICDE*, page 106, 2006.

[28] P. Hart, N. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Trans. SSC*, 4(2):100–107, 1968.

[29] Huahai He and Ambuj K. Singh. Closure-tree: an index structure for graph queries. In *ICDE*, pages 38–38, 2006.

[30] H. Hu, X. Yan, Y. Huang, J. Han, and X. J. Zhou. Mining coherent dense subgraphs across massive biological network for functional discovery. *Bioinfomatics*, 1(1):1–9, 2005.

[31] Derek Justice. A binary linear programming formulation of the graph edit distance. *IEEE TPAMI*, 28(8):1200–1214, 2006.

[32] M. Kanehisa and P. Bork. Bioinformatics in the post-sequence era. *Nature Genetics*, 33:305–310, 2003.

[33] H. W. Kugn. The hungarian method for the assignment problem. *Naval Research Logistics*, 2:83–97, 1955.

[34] Michihiro Kuramochi and George Karypis. Frequent subgraph discovery. In *ICDM*, pages 313–320, 2001.

[35] Chen Li, Jiaheng Lu, and Yiming Lu. Efficient merging and filtering algorithms for approximate string searches. In *ICDE*, pages 257–266, 2008.

[36] Chen Li, Bin Wang, and Xiaochun Yang. Vgram: improving performance of approximate queries on string collections using variable-length grams. In *VLDB*, pages 303–314, 2007.

[37] G. Li, X. Liu, J. Feng, and L. Zhou. Efficient similarity search for tree-structured data. In *SSDBM*, pages 131–149, 2008.

[38] Guoliang Li, Dong Deng, Jiannan Wang, and Jianhua Feng. Pass-join: a partition-based method for similarity joins. *PVLDB*, 5(3):253–264, 2011.

[39] W. J. Masek and M. Paterson. A faster algorithm computing string edit distance. *JCSS*, 20(1):18–31, 1980.

[40] Hugh McGuire and Brian O'Leary. *Book - A Futurist's Manifesto: a Collection of Essays from the Bleeding Edge of Publishing.* O'Reilly, 2012.

[41] Sergey Melink, Sriram Raghavan, Beverly Yang, and Hector Garcia-Molina. Building a distributed full-text index for the web. *ACM Trans. Inf. Syst.*, 19(3):217–241, 2001.

[42] Bruno T. Messmer and Horst Bunke. A new algorithm for error-tolerant subgraph isomorphism detection. *IEEE TPAMI*, 20(5):493–504, 1998.

[43] Krisztián Monostori, Arkdy Zaslavsky, and Heinz Schmidt. Document overlap detection system for distributed digital libraries. In *ACM DL*, pages 226–227, 2000.

[44] G. Navarro. A guided tour to approximate string matching. *ACM Comput. Surv.*, 33(1):31–88, 2001.

[45] G. Navarro, R. A. Baeza-Yates, E. Sutinen, and J. Tarhio. Indexing methods for approximate string matching. *IEEE Data Eng. Bull.*, 24(4):19–27, 2001.

[46] Michel Neuhaus, Kaspar Riesen, and Horst Bunke. Fast suboptimal algorithms for the computation of graph edit distance. In *SSSPR*, pages 163–172, 2006.

[47] R. Prasad and S. Agarwal. Study of bit-parallel approximate parameterized string matching algorithms. In *CCIS*, pages 26–36, 2009.

[48] Jianbin Qin, Wei Wang, Yifei Lu, Chuan Xiao, and Xuemin Lin. Efficient exact edit similarity query processing with the asymmetric signature scheme. In *SIGMOD*, pages 1033–1044, 2011.

[49] Vasanthakumar S. R., James P. Callan, and W. Bruce Croft. Integrating inquery with an rdbms to support text retrieval. In *Bull. Techn. Comm. Data Eng.*, pages 24–33, 1996.

[50] Kaspar Riesen and Horst Bunke. Iam graph database repository for graph based pattern recognition and machine learning. In *SSPR & SPR*, pages 287–297, 2008.

[51] Chuitian Rong, Wei Lu, Xiaoli Wang, Xiaoyong Du, Yueguo Chen, and Anthony K.H. Tung. Efficient and scalable processing of string similarity join. *IEEE TKDE*, 99, 2012.

[52] Adam Schenker. *Graph-theoretic techniques for web content mining.* PhD thesis, University of South Florida, 2003.

[53] B. A. Shapiro and K. Zhang. Comparing multiple rna secondary structures using tree comparisons. *Comput. Applic.in the Biosci.*, 6(4):309–318, 1990.

[54] Michael Stonebraker, Heidi Stettner, Nadene Lynn, Joseph Kalash, and Antonin Guttman. Document processing in a relational database system. *ACM Trans. Inf. Syst.*, 1(2):143–158, 1983.

[55] Zhan Su, Byung-Ryul Ahn, Ki-Yol Eom, Min-Koo Kang, Jin-Pyung Kim, and Moon-Kyun Kim. Plagiarism detection using the levenshtein distance and smith-waterman algorithm. In *ICICIC*, 2008.

[56] Erkki Sutinen and Jorma Tarhio. Filtration with q-samples in approximate string matching. In *CPM*, pages 50–63, 1996.

[57] Martin Theobald, Ralf Schenkel, and Gerhard Weikum. Efficient and self-tuning incremental query expansion for top-k query processing. In *SIGIR*, pages 242–249, 2005.

[58] Y. Tian and J.M. Patel. Tale: A tool for approximate large graph matching. In *ICDE*. IEEE, 2008.

[59] Anthony Tomasic, Héctor García-Molina, and Kurt Shoens. Incremental updates of inverted lists for text document retrieval. In *SIGMOD*, pages 289–300, 1994.

[60] Ismail H. Toroslu and Göktürk íçoluk. Incremental assignment problem. *Inf. Sci.*, 177(6):1523–1529, 2007.

[61] Ukkonen and Esko. Approximate string-matching with q-grams and maximal matches. *Theor. Comput. Sci.*, 92:191–211, 1992.

[62] Rares Vernica and Chen Li. Efficient top-k algorithms for fuzzy search in string collections. In *KEYS*, pages 9–14, 2009.

[63] Guoren Wang, Bin Wang, Xiaochun Yang, and Ge Yu. Efficiently indexing large sparse graphs for similarity search. *IEEE TKDE*, 99(PrePrints), 2010.

[64] Jiannan Wang, Jianhua Feng, and Guoliang Li. Trie-join: efficient trie-based string similarity joins with edit-distance constraints. *PVLDB*, 3(1-2):1219–1230, 2010.

[65] Jiannan Wang, Guoliang Li, and Jianhua Feng. Can we beat the prefix filtering? an adaptive framework for similarity join and search. In *SIGMOD*, 2012.

[66] Wei Wang, Chuan Xiao, Xuemin Lin, and Chengqi Zhang. Efficient approximate entity extraction with edit distance constraints. In *SIGMOD*, pages 759–770, 2009.

[67] Xiaoli Wang, Xiaofeng Ding, Anthony K. H. Tung, Shanshan Ying, and Hai Jin. An efficient graph indexing method. In *ICDE*, 2012.

[68] Xiaoli Wang, Xiaofeng Ding, Anthony K. H. Tung, and Zhenjie Zhang. Efficient and effective knn sequence search with approximate n-grams. *PVLDB*, 7, 2014.

[69] S. Wu, U. Manber, and E. W. Myers. A subquadratic algorithm for approximate limited expression matching. *Algorithmica*, 15(1):50–67, 1996.

[70] Xifeng Yan, Philip S. Yu, and Jiawei Han. Graph indexing: a frequent structure-based approach. In *SIGMOD*, pages 335–346, 2004.

[71] Xifeng Yan, Philip S. Yu, and Jiawei Han. Substructure similarity search in graph databases. In *SIGMOD*, pages 766–777, 2005.

[72] R. Yang, P. Kalnis, and A. K. H. Tung. Similarity evaluation on tree-structured data. In *SIGMOD*, pages 754–765, 2005.

[73] Xiaochun Yang, Bin Wang, and Chen Li. Cost-based variable-length-gram selection for string collections to support approximate queries efficiently. In *SIGMOD*, pages 353–364, 2008.

[74] Zhenglu Yang, Jianjun Yu, and Masaru Kitsuregawa. Fast algorithms for top-k approximate string matching. In *AAAI*, 2010.

[75] Zhiping Zeng, Anthony K. H. Tung, Jianyong Wang, Jianhua Feng, and Lizhu Zhou. Comparing stars: on approximating graph edit distance. *PVLDB*, 2(1):25–36, 2009.

[76] C. Zhang, J. Naughton, D. DeWitt, Q. Luo, and G. Lohman. On supporting containment queries in relational database management systems. In *SIGMOD*, pages 425–436. ACM, 2001.

[77] K. Zhang and D. Shasha. Simple fast algorithms for the editing distance between trees and related problems. *SIAM J. Comput.*, 18(6):1245–1262, 1989.

[78] Shijie Zhang, Meng Hu, and Jiong Yang. Treepi: a novel graph indexing method. In *ICDE*, pages 966–975, 2007.

[79] Zhenjie Zhang, Marios Hadjieleftheriou, Beng Chin Ooi, and Divesh S-rivastava. Bed-tree: an all-purpose index structure for string similarity search based on edit distance. In *SIGMOD*, pages 915–926, 2010.

[80] P. Zhao, J. Xu Yu, and P. S. Yu. Graph indexing: tree + delta >= graph. In *VLDB*, pages 938–949, 2007.

[81] Manuel Zini, Marco Fabbri, Massimo Moneglia, and Alessandro Panunzi. Plagiarism detection through multilevel text comparison. In *AXMEDIS*, pages 181–185, 2006.

[82] Justin Zobel and Alistair Moffat. Inverted files for text search engines. *ACM Comput. Surv.*, 38(2), 2006.