Ph.D. DISSERTATION

Similarity Query Processing Techniques for Text Data

텍스트 데이터에 대한 유사도질의 처리기술

BY

Kim, Younghoon

February 2013

SCHOOL OF ELECTRICAL ENGINEERING AND
COMPUTER SCIENCE
COLLEGE OF ENGINEERING
SEOUL NATIONAL UNIVERSITY

텍스트 데이터에 대한 유사도질의 처리기술

지도교수 심 규 석

위 논문을 공학 박사 학위논문으로 제출함

2012년 11월

서울대학교 대학원
전기컴퓨터공학부
김 영 훈

김영훈의 박사 학위논문을 인준함

2013년 1월

위 원 장 김 형 주 (인)

부위원장 심 규 석 (인)

위 원 이 상 구 (인)

위 원 홍 성 수 (인)

위 원 김 철 연 (인)

Abstract

With the widespread use of the internet, text-based data sources have become ubiquitous and the demand for effective support of similarity queries in text data continues to increase. While the applications for text similarity queries are diverse, similarity queries are essential and useful in those applications. The traditional approach for similarity queries is to find every similar document satisfying a given minimum similarity threshold which is required to be provided by each user. However, since users have to try repeatedly different minimum similarity threshold to find a proper value, the schemes for top-k similarity query processing are more practical and have been extensively explored recently.

In this dissertation, we first introduce motivating applications utilizing the text similarity queries which are practically applied in many applications including commercial search engines such as Google, Bing and Yahoo!. The useful similarity queries usually consist of exact substring matching, top-k approximate substring matching and top-k approximate string joins. Thus, we investigate the problem of processing these similarity queries efficiently.

For exact substring matching, we propose the optimal algorithm to find the best query plan utilizing inverted variable-length gram indexes. We also develop the approximate algorithms to overcome the exponential search space for finding an optimal plan. For top-k approximate substring matching, we propose effective filtering techniques utilizing our novel lower bounds for substring edit distance. Then, we develop efficient algorithms for top-k approximate substring matching applying our filtering techniques. For top-k approximate string joins, we devise efficient

algorithms as well as the essential pair partitioning technique which allows us to effectively ignore many pairs of strings in advance. Our experiments show that the proposed algorithms reduce the query processing cost very efficiently.

Since the similarity queries studied in this dissertation are widely required in many applications such as web search engines, we believe that our proposed algorithms will enhance the performance of those applications practically.

Keywords: *Similarity queries, exact substring matching, top-k approximate substring matching, top-k approximate string joins*

Contents

List of Figures

Chapter 1

Introduction

With the widespread use of the Internet, text-based data sources have become ubiquitous and the demand for efficient query processing for text data continues to increase. A list of possible applications includes keyword matching for web search[3], short snippet suggestions[69], named entity recognition[85], finding DNA subsequences for bio-informatics[61], music data retrieval[55], spelling suggestion[84], duplicated document detection[93] and document clustering[100]. These applications generally require high performance for real-time similarity query processing. While these applications are not all new, as there is an increasing trend of applications to deal with vast amounts of data, similarity query processing for text data becomes a more challenging problem today.

Similarity queries used widely on text data include *similarity matching* and *similarity join*. Given a set of documents and a query string, the *similarity matching* is to find every document which includes substrings similar to the query string with respect to a given minimum similarity threshold. On the other hand, the *similarity join* is to discover every pair of documents in the set of documents which are similar

to each other with respect to a given minimum similarity threshold.

In traditional approaches, similarity queries require a minimum similarity threshold to retrieve all documents satisfying the given minimum similarity threshold. However, it is very difficult to know a proper threshold before querying the database actually. Thus, a user has to try different similarity threshold values, which may lead to empty results (if the threshold chosen is too high) or too many results with a long running time (if the threshold is too low). An appealing alternative method is to provide a user the most similar k documents without requesting the user to specify a minimum similarity threshold. Actually, the top-k similarity queries are preferred in many application. In the following, we present some relevant applications which require such text similarity queries.

Keyword matching in web search engines: Every web search engine utilizes the similarity matching queries in order to find web pages containing the query string or similar strings to the query string[104, 3]. Search engines generally show the top-k similar query results only without asking each user a minimum similarity threshold.

Spelling suggestion: Recognizing misspelled words and suggesting the correct spellings are very useful facilities in many useful applications[84]. The suggestions of the correct spellings are usually done by discovering top-k most similar strings in a dictionary to a given query string.

Named entity recognition: Named entity recognition is the technique to discover named entities from unstructured text data. A typical approach is to find similar substrings in the text data with respect to each entity in a given dictionary, which includes such as the names of famous people in history or the names of places in the world[85, 103].

Near duplicate document detection and elimination: Detecting the duplicated documents and removing the identical (or very similar) documents is very important to enhance web search engines[6]. In many cases, every pair of duplicated documents are discovered using similarity join queries before indexing the documents, and only the representative documents are indexed in the web search engines.

While the applications of text query processing are diverse, similarity queries are essential and useful in those applications including commercial search engines such as Google[29], Bing[11] and Yahoo![95]. We next present motivating applications where our similarity query processing techniques studied in this dissertation can be used effectively.

1.1 Motivating Applications of Similarity Text Queries

Let us consider a web-based search engine system which indexes a database storing the text messages posted in Twitter[82]. Users may be interested in retrieving the messages with useful information by submitting query strings to the search engine. The following are the scenarios of the usages of the search engine.

Exact substring matching: For a query string, the search engine retrieves all messages containing the query string. We call such a similarity matching to find the documents including the query string *exact substring matching*.

For example, consider the Twitter messages stored in a relational table with the schema *twitter(id, username, message)*, where *id* is the primary key of record, *username* is the name of user who writes the message, and *message* is the short text message. We present the table 'twitter' in Figure 1.1(a).

id	username	message
1	BestArtists	Paul Jackson Pollock (January 28, 1912 – August 11, 1956), known as Jackson Pollock, was an influential American painter and a major figure in the abstract ... his painting is ...
2	Shelly	You are the next Michael jackson, you pedofile @50cent
3	Oscar Torres	Jackson Pollock @ Los Angles Country Museum of Art (LACMA) http://instagr.am/SJaue49rjf
4	Tom Ben	Modern art, like a striped down Jackson Pollack painting... http://did.bz/K48912D
5	Stephanie Keith	A small Jackson Pollack painting sold for more than 30 million. He is the greatest modern artist in the 19-th century.
6	V. Bridges Hoyt	"Every good painter paints what he is" - Jackson Pollock

(a) A table with a scheme *twitter(id,username,message)*

title	text
Jackson Pollock	Paul *Jackson Pollock* (January 28, 1912 – August 11, 1956), known as Jackson Pollock, was an influential *American painter* and a major figure in the abstract ... his *painting* is ...
Museum of Modern Art	It also holds works by a wide range of influential European and *American artists* including ... *Jackson Pollock*, ... Picasso's *painting* Guernica (on loan to MoMA at the time) the *painting* depicts.

(b) A table with a scheme *wikipedia(title,text)*

Figure 1.1: The tables 'twitter' and 'wikipedia'

Assume that a user wants to find Twitter messages mentioning the artist 'Jackson Pollock'. The following SQL query is to obtain the Twitter messages including the query string.

```
SELECT *
FROM twitter
WHERE message LIKE '%Jackson Pollock%'
```

By querying with the above SQL query, the user can obtain the query result of messages containing the query string as illustrated in Figure 1.2.

There may be no message at all containing the query string exactly, or the number of retrieved messages may not be enough to find useful information for which the user is searching. For example, the user might have misspelled the artist name.

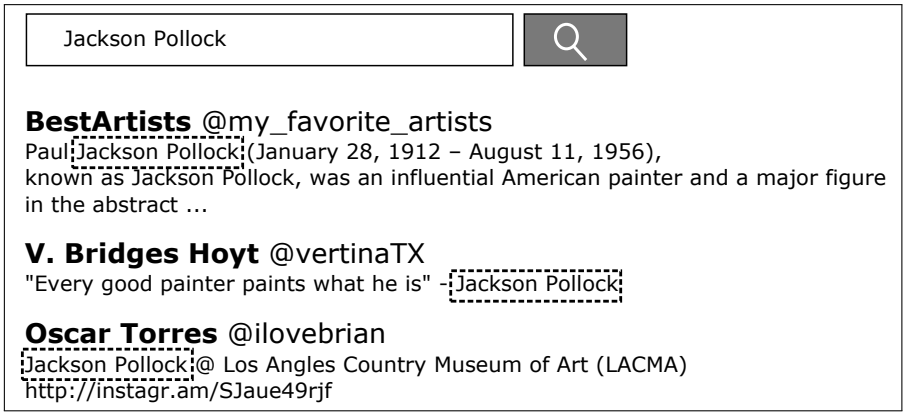


Figure 1.2: Twitter messages containing the query string 'Jackson Pollock' exactly

Furthermore, it is also possible that the useful messages which the user is searching for are not retrieved simply because the authors of the Twitter messages misspelled the artist name in their Twitter messages. In such cases, the following query would be very useful.

Top-k approximate substring matching: By finding Twitter messages which contain strings similar to the query string, we can provide users more useful information which cannot be obtained by the capability of the exact substring matching only. For instance, even when users type the misspelled name 'Jackson Polock', it will be very useful if the search engine finds the messages mentioning 'Jackson Pollock' correctly. Furthermore, it will be also useful if each user can correct his query string interactively with being consulted by the query strings suggested automatically in the search engine. We refer to such a similarity matching, which finds the top-k documents including the most similar strings to the query string, as *approximate substring matching*.

Suppose that a user typed a misspelled query string 'Jackson Polock'. Let $sim(s_1, s_2)$

Figure 1.3: Twitter messages containing the substrings similar to 'Jackson Polock'

denote a user-defined function in a relational database which computes the similarity between two strings s_1 and s_2. Given a table 'twitter' in Figure 1.1, the following SQL sentence represents the top-k approximate substring matching query for the query string 'Jackson Polock':

```
SELECT *
FROM twitter
ORDER BY sim(message,'Jackson Polock') DESC
LIMIT k
```

Assume the we also have a table with a scheme *wikipedia(title,name)*, where *title* and *text* are the title and the contents of Wikipedia page [86], as shown in Figure 1.1(b). Regarding the titles of Wikipedia page as the correct words, the search engine can suggest the correct query strings with the following SQL query:

```
SELECT *
FROM wikipedia
```

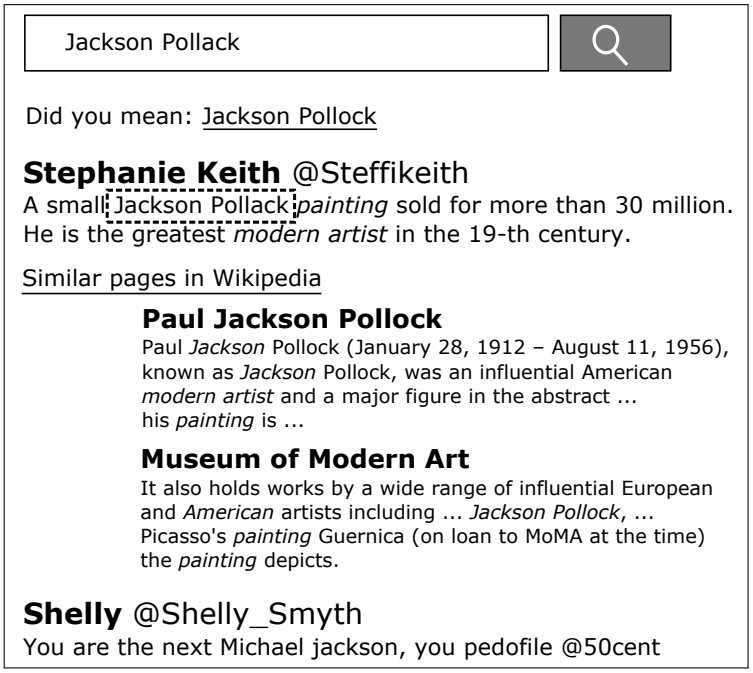


Figure 1.4: Twitter messages and Wikipedia pages for the query 'Jackson Pollack'

```
ORDER BY sim(title,'Jackson Polock') DESC
LIMIT k
```

Figure 1.3 shows the top-k messages containing the approximate strings of the query string 'Jackson Pollock' as well as the correct words suggested for the misspelled query string.

Despite of using the exact and approximate substring matching, if the user cannot discover a satisfactory query result, he may want to browse more documents, which are similar to the messages retrieved by the substring matching queries, from other databases.

Top-k approximate string join: For the messages discovered by the substring

matching, the search engine can help users to find useful information by providing the similar documents to the messages found from other data sources. For example, suppose that a user wants to find the biography of the artist 'Jackson Pollock' but does not know the exact spell of his name. For the Twitter messages retrieved by the substring matching, if the search engine shows the similar documents found from Wikipedia together, it will very useful for the user to discover the biography of the artist as illustrated in Figure 1.4. We refer to the query processing to find the k most similar pairs of documents as *top-k approximate string join*.

Given the tables 'twitter' and 'wikipedia' in Figure 1.1, the search engine can generate the query result in Figure 1.4 using the following SQL query:

```
SELECT t1.*, t2.*
FROM
  (SELECT * FROM twitter
   WHERE message LIKE '%Jackson Pollock%' LIMIT k1)
AS t1,
  (SELECT * FROM wikipedia
   ORDER BY sim(text,'Jackson Pollock') DESC LIMIT k2)
AS t2
ORDER BY sim(t1.message,t2.text) DESC
LIMIT k3
```

To utilize these useful and practical similarity queries in web search engines, we need efficient query processing algorithms. For example, consider the above SQL join query which includes two nested selection queries of top-k approximate substring matching. Intuitively, the query optimization will perform each nested query and then next execute the join query. Thus, we are required to optimize the

processing not only for the nested queries of top-k approximate substring matching, but also for the top-k approximate join between the query results of the nested queries.

There have been studied many query optimization algorithms in relational databases. For similarity text queries, selectivity estimation of string predicates with edit distance was studied and utilized in finding optimal query plans[36, 47]. For example, to estimate the result size of approximate substring matching, Jin and Li [36] proposed an estimation algorithm using the clusters of similar strings. Lee, Ng and Shim in [47] estimate the result size of approximate substring matching by using the extended q-grams with wildcards. However, they focused on the estimation of query result sizes to utilize the estimated size in query optimization and does not provide actual query processing algorithms for approximate substring matching. Furthermore, the traditional similarity query processing algorithms for text data are mainly focused on the similarity queries with a given minimum similarity threshold[63, 62, 8, 52, 54]. Thus, in this dissertation, we address and develop new techniques which enhance the similarity query processing on text data.

1.2 Contributions of This Dissertation

In this dissertation, we propose efficient query processing algorithms for the similarity queries discussed in our motivating applications. Our contributions are as follows:

- We first study the optimization of query processing for *exact substring matching* by taking advantage of the inverted indexes using variable-length grams [53].

We propose the optimal algorithms to find the best plan for an exact substring matching query, which minimizes the query processing cost. We also provide the approximate algorithms to overcome the exponential nature of the search space required to be explored for finding optimal an query plan.

- We next investigate the top-k approximate substring matching problem. To the best of our knowledge, no existing work has addressed the top-k approximate substring matching problem and our algorithms presented in this dissertation are the first work for the problem. Using q-grams, we first propose our novel lower bounds of the edit distances between a query string and all substrings appearing in a string. Then, we present our efficient top-k approximate substring matching algorithms which reduce query processing cost effectively.

- We finally present our efficient top-k approximate string join algorithms. We propose the divide-and-conquer and branch-and-bound algorithms which compute the top-k similar pairs of documents efficiently. Then, we present a partitioning method which allows us to ignore the most of document pairs which cannot not participate the top-k similar pairs. Utilizing the partitioning method, we can decrease query processing costs dramatically in the top-k approximate string joins.

Since the similarity queries studied in this dissertation are widely required in many applications such as web search engines, our proposed algorithms will enhance the performance of those applications practically.

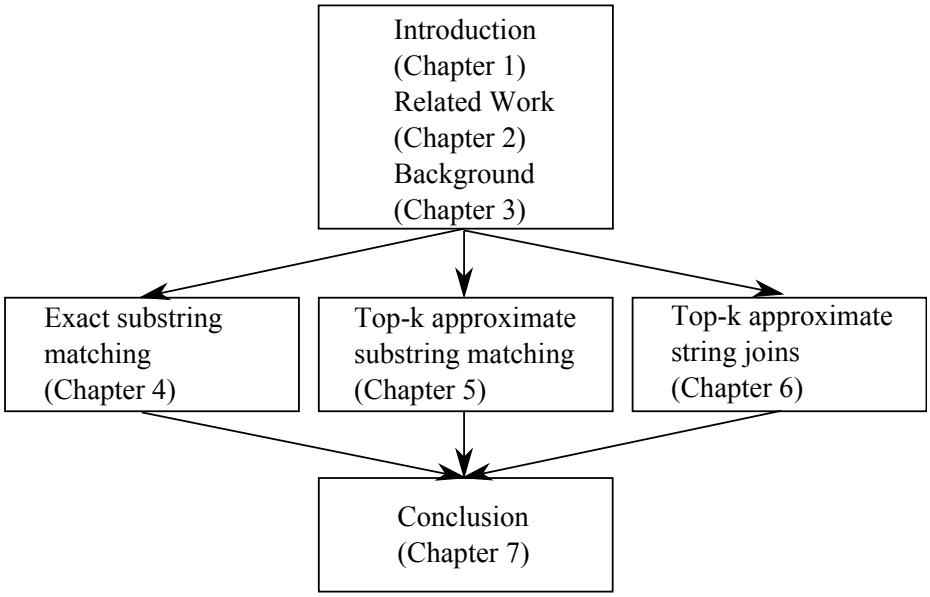


Figure 1.5: Dissertation overview

1.3 Dissertation Overview

The remaining chapters of this dissertation is organized as follows. In Chapter 2, we review the previous works on similarity query processing. In Chapter 3, we provide the background to this dissertation which is commonly utilized in the remaining chapters. In Chapter 4, we study the problem of exact substring matching and provide efficient substring matching algorithms using inverted index with variable length grams. In Chapter 5, we investigate the problem of finding top-k approximate substring matches and provide efficient top-k approximate substring matching algorithms using our proposed filtering methods. In Chapter 6, we examine the problem of top-k approximate string joins. We finally present our conclusions and future work in Chapter 7. The diagram in Figure 1.5 outlines how the chapters of this dissertation are related to each other.

Chapter 2

Related Work

In this chapter, we present related work on similarity query processing for text data. We first discuss on the distance measures used for similarity text queries and next study the previous work of similarity query processing techniques.

To handle similarity queries in text data, various string (dis)similarity measures, such as edit distance, hamming distance, Jaccard coefficient and cosine similarity have been considered[4, 37, 44, 58]. These distance measures are the most widely accepted distance measures for database applications where domain specific knowledge is not really available[37, 61, 85]. Among them, edit distance is the most popular distance measure to search the approximate substring matches for a given query string. Furthermore, to search the similar strings to a given query string, Euclidean distance and cosine similarity are frequently used[58, 92].

Similarity queries can be classified into two categories: threshold and top-k queries, which are known as the basic types for similarity queries[98]. We will next review the previous works on the query processing techniques for each type of similarity queries.

	Threshold queries	Top-k queries
String matching	[1], [8], [47], [36], [54] [52], [66], [85]	[97]
String joins	[7], [6], [16], [39], [102], [73], [91], [99]	[94], [99]

Figure 2.1: The related works on similarity queries

2.1 Similarity Threshold Queries

Traditionally, the most common type of a similarity query is the similarity threshold query. The query is specified by a query string σ with a minimum similarity threshold θ and then all strings satisfying the similarity threshold θ are retrieved. Frequently, instead of the minimum similarity threshold, users are required to provide the maximum distance threshold τ to find every string whose distance to the query string σ is at most τ.

Approximate string matching: For similarity string matching with a maximum distance threshold τ, the algorithms using inverted q-gram indexes are presented in [8] and [52] which generalize the list merging method in [74]. However, in [85], it is shown that it is not always possible to use existing inverted q-gram indexes for approximate string matching. In the paper, they proved that, in order to utilize inverted q-gram indexes, the length of q-grams to be used should be smaller than $(\min(L_{min}, |\sigma|)+1)/(\tau+1)$ where L_{min} is the smallest string size among all strings in D.

String matching to find documents including a given query string exactly has been also extensively studied including our previous work in [42]. For exact substring matching, on the contrary to approximate substring matching, existing in-

verted q-gram indexes can be utilized without such a constraint in [85] and thus we could develop efficient algorithms using inverted q-gram indexes.

To speed up the processing of similarity string matching queries, the techniques to find the strings satisfying the minimum similarity threshold (or the maximum distance threshold) approximately are also developed. In [1] and [66], they propose the algorithms which find similar strings probabilistically resulting that some similar strings satisfying the threshold may be missed.

Similarity string matching has been studied in the context of approximate entity extraction in [54] and [85]. With a given threshold, these algorithms find every substring such that the edit distance between the substring and a string in a given entity dictionary is at most the threshold.

The selectivity estimation for approximate string matching queries was also studied [36, 47]. In [36], an estimation algorithm which builds clusters of similar strings and maintains histograms that store the distribution information of strings with augmented edit distance information. The extension of q-grams with wildcards for estimating selectivity of string matching with edit distance was also proposed in [47].

Approximate string joins: The similarity join for text data has been studied for various applications, such similar name detecting[10] and data cleansing[73]. The similarity join algorithms with a given threshold using inverted indexes for pruning were proposed in [7], [16] and [73].

In [6], to develop a self join algorithm for finding similar pairs of vectors using cosine similarity, the self-join framework which performs the index building and similarity join processing is proposed. This idea is further exploited for finding the pairs of similar strings from a set of strings in [91]. Furthermore, the algorithms

in [99] utilize suffix tries which index a small portion of suffixes only in the data. While exploring the suffix tries and perform joins with every pair of nodes, they prune the branches in the suffix tries not containing string candidates by computing the lower bound of distances. However, to use suffix tries for approximate substring matching, we have to index every suffix appearing in the data, resulting large index sizes. Because of the high space requirements and poor locality of suffix tries, it is mentioned in [64] that the suffix trie based approach works well only when both of the data and index fit in main memory.

To compute similar pairs of documents, Euclidean distance and cosine similarity were also frequently used by representing documents in the vector space model[71]. The algorithms in [39] and [102] were proposed to find the top-k similar pairs for two dimensional data using Euclidean distance. The work in [39] iteratively reduces the search space using Voronoi diagrams. In [102], the authors investigated the use of R-tree indexes for efficient pruning. However, since the time complexity of the algorithms grow exponentially with d, they are not suitable for approximate joins for text data.

2.2 Similarity Top-k Queries

As we mentioned in Chapter 1, since it is very difficult for each user to know a proper threshold in advance, it is more practical to find the most similar k documents without specifying a minimum similarity threshold. A similarity top-k query consists of a query string σ and the number of strings k to retrieve. A sequence of strings in the set of strings, which contains the top-k similar strings, is returned for a similarity top-k query with the decreasing order of their similaries to the query string σ. When

we use a distance measure instead of similarity measures, the query retrieves the top-k strings in the increasing order of the distances to σ.

Top-k approximate string matching: To the best of our knowledge, no existing work addresses the top-k approximate substring matching problem correctly and our algorithms presented in this dissertation are the first work for the problem. A top-k approximate *string* matching algorithm using inverted q-gram indexes and the count filtering was proposed in [97]. However, their algorithm cannot find the top-k approximate string matches correctly because we cannot always ensure that the length of q-grams used must be smaller than $(|\sigma| + 1)/(\tau_k + 1)$ when τ_k is the k-th smallest substring edit distance in D, as mentioned in [85].

Top-k approximate string joins: In [99], they proposed a top-k string join algorithm utilizing suffix tries. However, the algorithm has exponential space and time complexities with the threshold τ since it has to compute similarity between the every possible pair of nodes in the suffix trie. Furthermore, it works only when the suffix trie fits in the main memory.

To process top-k simiarity joins using cosine similarity measure in vector space, the similarity top-k query processing algorithm for Jaccard coefficient in [94] was extended to work with other similarity (distance) measures such as overlap similarity and cosine distance.

Chapter 3

Background

In this chapter, we provide the technical background commonly used in this dissertation. We first present the definitions of q-grams, variable-length gram and inverted indexes, which are used for the query processing of exact and approximate substring matching. We next introduce two distance measures between two strings, which are the edit distance [50] and substring edit distance [76], as well as the dynamic programming algorithms to compute the distances. Then, we describe the MinHash technique [101] which is used when we estimate the size of query result to find optimal query plans in our works.

3.1 Substring Matching with Inverted q-gram Indexes

Inverted indexes are the most widely used data structures for similarity query processing [14, 15, 19, 31, 65, 40, 81, 101, 104]. In this section, we first present how to process substring matching queries using the inverted q-gram indexes.

q-gram and inverted q-gram index: We use R, possibly with subscripts, to denote tables. Let Σ be a finite alphabet of size $|\Sigma|$. We use σ to denote strings in Σ^*. Let

$\sigma \in \Sigma^*$ be a string of length n. We use $\sigma[i, j]$, $1 \leq i \leq j \leq n$, to denote a substring of σ of length j-i+1 starting from the position i.

Definition 3.1.1 The q-grams of a string σ are $\sigma[i, i + q - 1]$ with $1 \leq i \leq |\sigma| - q + 1$. For a set of q-grams of σ, if the character in every location of σ is covered by at least a q-gram in the q-gram set, we say that the q-gram set *covers* the string σ and the set is called a *covering q-gram set* of σ. The *maximal and minimal covering q-gram sets* for a given string σ are the covering q-gram sets with the maximum and minimum sizes respectively.

The q-grams are obtained by sliding a window of length q over the characters of σ. The sizes of the maximal and minimal covering sets of a string σ are $|\sigma|$-q+1 and $\lceil |\sigma|/q \rceil$ respectively.

Example 3.1.2 For the string 'student', the maximal and minimal covering 3-gram sets are {'stu', 'tud', 'ude', 'den', 'ent'} and {'stu', 'den', 'ent'} respectively. {'stu', 'ude', 'ent'} is also a minimal covering 3-gram set. ∎

Definition 3.1.3 The positional q-grams of a string σ are ($\sigma[i, i + q - 1]$, i) with $1 \leq i \leq |\sigma| - q + 1$. We can define the maximal and minimal covering positional q-gram sets similarly for positional q-gram sets.

Example 3.1.4 For the string 'student', {('stu',1), ('ude',3), ('ent',5)} is a minimal covering positional 3-gram set. ∎

Given a q-gram, a pair of record ID and the position where the q-gram occurs in the record is called *posting*. The posting list of a q-gram contains the list of postings for every occurrence of the term in all records.

gram	Posting list
sub	(r_2,1)
ubs	(r_2,2)
bst	(r_2,3)
str	(r_1,1), (r_2,4)
tri	(r_1,2), (r_2,5)
rin	(r_1,3), (r_2,6)
ing	(r_1,4), (r_2,7)

Figure 3.1: An example of inverted q-gram index

The *inverted index* is an index which can quickly search the records containing a given term for the indexed attribute. For each term of every string in the indexed attribute, its postings are inserted into the term's posting list in the inverted index. A frequently used definition of the term is the q-gram. We call such an index an *inverted q-gram index*. An example of an inverted 3-gram index for the relation {(r_1, 'string'), (r_2, 'substring')} was previously shown in Figure 3.1.

vq-gram and inverted vq-gram indexes: We can also consider the case where an index is built with variable length grams whose definition below was borrowed from [53].

Definition 3.1.5 For a string σ, the variable-length grams with q_{min} and q_{max} ($q_{min} \leq q_{max}$) are all substrings from σ whose lengths are between q_{min} and q_{max}. We will also represent variable-length grams as *vq-grams*.

Example 3.1.6 With $q_{min} = 4$ and $q_{max} = 5$, all possible variable-length grams of the string 'substring' are {'subs', 'ubst', 'bstr', 'stri', 'trin', 'ring', 'subst', 'ubstr', 'bstri', 'strin', 'tring'}. ∎

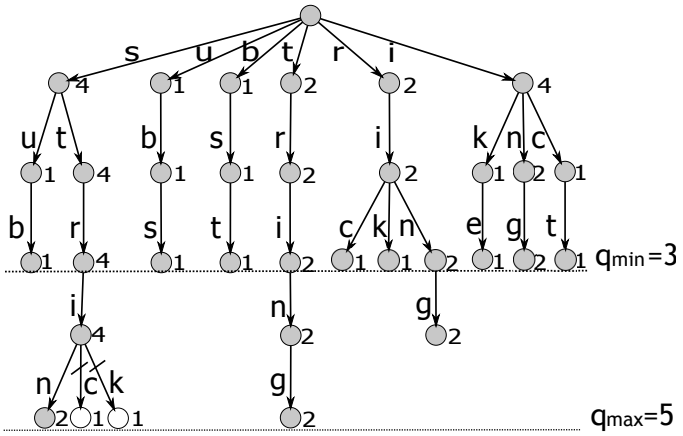


Figure 3.2: vq-gram dictionary

Given a collection of strings, a *vq-gram dictionary* with q_{min} and q_{max} is a subset of all possible vq-grams of lengths between q_{min} and q_{max} from the collection. When we build a vq-gram dictionary, we may use a subset of strings in the table to be indexed. An inverted vq-gram index is built with a preexisting vq-gram dictionary and each vq-gram in the dictionary becomes a key in the inverted index if the vq-gram is contained in a string to be indexed.

For building a vq-gram dictionary, we use the algorithm *Prune* proposed in [53], which is designed to perform pruning of vq-grams whose frequencies are too small, since the index size will be very large if the vq-gram dictionary contains all possible vq-grams. Given a collection of strings, it first builds a trie [43] with all possible vq-grams whose lengths are between q_{min} and q_{max}. Every path from the root to each node in the trie represents a vq-gram, and each node has the frequency of the vq-gram in the collection of strings. Then, for every node whose depth is larger than q_{min} in the trie, the algorithm performs pruning of a subset of child nodes

whose sum of frequencies is maximal but not larger than the given pruning threshold. For example, Figure 3.2 shows a vq-gram dictionary built with a collection of strings {'substring', 'string', 'strike', 'strict'}, where q_{min}=3, q_{max}=5, and the pruning threshold is 2. At the node of 'stri', the child nodes of 'stric' and 'strik' are pruned by the algorithm, because the sum of their frequencies is not larger than 2.

Substring matching with inverted indexes: We next discuss how to evaluate queries with an inverted index using vq-grams. When we are given a query string σ and a preexisting vq-gram dictionary D, we first generate positional vq-grams, which are $Q = \{(g_1,p_1), (g_2,p_2), \cdots, (g_n, p_n)\}$, from σ that appear in D and retrieve the corresponding posting list of each vq-gram from the inverted index. Then, we perform the intersection of all posting lists of vq-grams in Q and return the distinct record IDs.

For the positional vq-grams (g_1, p_1), ..., (g_n, p_n) and L which is the intersection result of the posting lists of g_1, ..., g_n, the set $\{(g_1, p_1),...,(g_n, p_n)\}$ is called the *identifier* of L and is represented by $I(L)$. For each posting list L, we let $p_{min}(L)$ be the smallest position appearing at $I(L)$, i.e. $\min_{(g_i,p_i)\in I(L)}\{p_i\}$. Let $\{(g_{i_1},p_{i_1}),...,(g_{i_n},p_{i_n})\}$ and $\{(g_{j_1},p_{j_1}),...,(g_{j_m},p_{j_m})\}$ be the identifiers of the posting lists L_i and L_j respectively. Then, we define the following two operations.

Definition 3.1.7 $L_i \hat{\bigcap} L_j$ represents the intersection result between two posting lists L_i and L_j where, for each pair of postings (r_k, x) and (r_k, y) from L_i and L_j respectively, $(r_k, \min(x, y))$ is included if (x-y) = ($p_{min}(L_i)$-$p_{min}(L_j)$).

Definition 3.1.8 For a given posting list L, let $\Pi_{rid}(L)$ denote the set of distinct record IDs in L.

gram	Posting list
sub	(r_2,1)
ubs	(r_2,2)
bst	(r_2,3)
str	(r_1,1), (r_2,4)
tri	(r_1,2), (r_2,5)
rin	(r_1,3), (r_2,6)
ing	(r_1,4), (r_2,7)
ubst	(r_2,2)
bstr	(r_2,3)
bstri	(r_2,3)

Figure 3.3: An example of inverted vq-gram index

Example 3.1.9 Consider a relation {(r_1, 'string'), (r_2, 'substring')} and the inverted vq-gram indexes for the relation shown in Figure 3.1. Suppose we have a query string 'string' whose positional vq-grams are {*(str, 1), (tri, 2), (ing, 4)*}. Let L_{str},L_{tri} and L_{ing} be the posting lists of str, tri and ing respectively. The identifiers of L_{str} and L_{ing} are {*('str', 1)*} and {*('ing', 4)* } respectively. Then we have $p_{min}(L_{str})$=1 and $p_{min}(L_{ing})$=4. Since the pair of postings (r_2, 4) in L_{str} and (r_2, 7) in L_{ing} satisfies (4-7) = $p_{min}(L_{str})$-$p_{min}(L_{ing})$, we put (r_2, 4) into the result list L. Similarly, for the pair of (r_1, 1) in L_{str} and (r_1, 4) in L_{ing}, we put (r_1, 1) into L. Thus, we have $L_{str}\hat{\bigcap}L_{ing}$= {($r_1$, 1), ($r_2$, 4)}. The identifier of L becomes {*('str', 1), ('ing', 4)*}. Thus, $p_{min}(L)$ becomes 1 and will be used later for the intersection with another posting list such as L_{tri}.

We will next perform the intersection between L and L_{tri}. Since L_{tri} = {(r_1,2), (r_2,5)} in Figure 4.1(a) and $p_{min}(L_{tri})$-$p_{min}(L)$=1, for the pair of (r_1,1) in L and (r_1,2) in L_{tri}, we put (r_1,1) to the result list L'=$L\hat{\bigcap}L_{tri}$. Similarly, for the pair of (r_2,4) and (r_2,5), we put (r_2, 4) to L' resulting L'=$L \hat{\bigcap} L_{tri}$={(r_1,1), (r_2,4)}. Thus, we have $\Pi_{rid}(L')$={r_1,r_2}. ∎

Instead of performing the intersection of two posting lists and storing the result list repeatedly, we can process a pipelined $|Q|$-way join for $|Q|$ posting lists in a positional vq-gram set Q not to incur I/O cost of writing the intermediate result to disk. Thus, throughout the rest of this dissertation, we assume that the pipelined $|Q|$-way join is performed for intersecting the posting lists of vq-grams in Q.

3.2 Similarity Measures between Strings

To handle similarity queries in text data, various string (dis)similarity measures, such as edit distance, hamming distance, Jaccard coefficient and cosine similarity have been considered [4, 37, 44, 58]. These distance measures are the most widely accepted distance measures for the database applications where domain specific knowledge is not really available [37, 47, 61, 85]. Among them, the edit distance is the most popular distance measure to search the approximate substring matches for a given query string. Furthermore, to find the documents containing words with similar distribution to the query string (or the other document), the Euclidean distance and cosine similarity are frequently used [92, 47, 58].

We next describe the edit distance [50] and substring edit distance [76] used in our works. We also provide two dynamic programming algorithms to calculate those distances respectively.

Edit distance: For two strings s_i and s_j, we assume that s_i can be transformed to s_j by applying repeatedly the three operations: insertion, deletion and substitution. Then, the edit distance $d(s_i, s_j)$ between two strings s_i and s_j is defined as follows [50]:

Definition 3.2.1 For two strings s_i and s_j, the *edit distance* between s_i and s_j is defined as the minimum number of operations needed to transform s_i to s_j (or s_j to s_i), denoted by $d(s_i, s_j)$.

The basic algorithm in [50] to compute the edit distance between two strings s_i and s_j is based on dynamic programming. Let $c(u, v)$ be the minimum number of operations to transform $s_i[1, u]$ to $s_j[1, v]$ with $1 \leq u \leq |s_i|$ and $1 \leq v \leq |s_j|$. We can convert $s_i[1, u]$ to $s_j[1, v]$ by performing one of the following transformations which has the minimum number of edit operations.

- When $s_i[u] = s_j[v]$, convert $s_i[1, u-1]$ to $s_j[1, v-1]$ with the minimum number of operations (i.e., $c(u-1, v-1)$)
- When $s_i[u] \neq s_j[v]$,
  - append $s_j[v]$ at the end of $s_i[1, u]$ and change $s_i[1, u]$ to $s_j[1, v-1]$ with the minimum number of operations (i.e., $c(u, v-1)$)
  - delete $s_i[u]$ and transform $s_i[1, u-1]$ to $s_j[1, v]$ with the minimum number of operations (i.e., $c(u, v-1)$)
  - replace $s_i[1, u]$ with $s_j[v)$ and convert $s_i[1, u-1]$ to $s_j[1, v-1]$ with the minimum number of operations (i.e., $c(u-1, v-1)$)

From the above, we can obtain the recursive formula for computing the edit distance by dynamic programming as follows:

$$\begin{aligned} c(u, v) = \min\{&c(u-1, v-1) + \delta(s_i[u], s_j[v]), \\ &c(u-1, v) + 1, c(u, v-1) + 1\}, \end{aligned} \tag{3.1}$$

		string											
		J	a	c	k	s	o	n	v	i	l	l	e
	0	1	2	3	4	5	6	7	8	9	10	11	12
J	1	0	1	2	3	4	5	6	7	8	9	10	11
a	2	1	0	1	2	3	4	5	6	7	8	9	10
c	3	2	1	0	1	2	3	4	5	6	7	8	9
k	4	3	2	1	0	1	2	3	4	5	6	7	8
i	5	4	3	2	1	1	2	3	4	4	5	6	7
e	6	5	4	3	2	2	2	3	4	5	5	6	6

query

Figure 3.4: Computation of the edit distance

where $\delta(c_1, c_2)$=0 if c_1=c_2 and $\delta(c_1, c_2)$=0 otherwise. Initially, $c(0, v) = v$ for every v with $0 \leq v \leq |s_j|$ and $c(u, 0) = u$ for every u with $0 \leq u \leq |s_i|$ since the transformation from an empty string to a string of length i requires at least i insertions. After computing $c(u, v)$ for every $1 \leq u \leq |s_i|$ and $1 \leq v \leq |s_j|$, $c(|s_i|, |s_j|)$ stores the edit distance between s_i and s_j (i.e., $d(s_i, s_j)$). Since we need to compute $c(u, v)$ with every $1 \leq u \leq |s_i|$ and $1 \leq v \leq |s_j|$, and computing each $c(u, v)$ takes $O(1)$ times, the time complexity of edit distance computation is $O(|s_i| \cdot |s_j|)$.

Example 3.2.2 Consider two strings s_1 = 'Jackie' and s_2 = 'Jacksonville'. In Figure 3.4, we show the values of $c(u, v)$s for computing the edit distance of s_1 and s_2. Initially, $c(0, v)$s with $0 \leq v \leq 12$ are set to v and every $c(u, 0)$ with $0 \leq u \leq 6$ is set to u. To compute $c(u, v)$, we need to have $c(u, v)$, $c(u-1, v)$ and $c(u, v-1)$ only. For example, to compute $c(5, 9)$ which is the edit distance between two substrings 'Jacki' and 'Jacksonvi', the substrings can be transformed to each other with $c(4, 8)$=4 because the last letter is the same. Finally, we obtain the edit distance which is $c(6, 12)$=6. ∎

Substring edit distance: The substring edit distance from a string s to another string σ, which is represented by $d_{sub}(s, \sigma)$, is the minimum among the edit distances

between σ and every substring of s. The $O(|\sigma||s|)$-time algorithm based on dynamic programming to compute the substring edit distance was introduced in [76].

For two strings σ and s, let $t(u, v)$ be the substring of s ending at the position v such that the edit distance between $\sigma[1, u]$ and the substring $t(u, v)$ is the minimum among the edit distances between $\sigma[1, u]$ and all substrings of s ending at the position v. Let $m(u, v)$ denote the edit distance between σ and $t(u, v)$. Note that $t(u, v)$ is defined conceptually and we do not maintain $t(u, v)$ actually. We can compute $m(u, v)$ by selecting one of the following transformations from σ to $t(u, v)$ with the minimum number of edit operations.

- When $\sigma[u] = s[v]$, convert $\sigma[1, u-1]$ to $t(u-1, v-1)$ with the minimum number of operations (i.e., $m(u-1, v-1)$)
- When $\sigma[u] \neq s[v]$,
  - append $s[v]$ at the end of $\sigma[1, u]$ and change $\sigma[1, u]$ to $t(u, v-1)$ with the minimum number of operations (i.e., $m(u, v-1)$)
  - delete $\sigma[u]$ and transform $\sigma[1, u-1]$ to $t(u-1, v)$ with the minimum number of operations (i.e., $m(u, v-1)$)
  - replace $\sigma[u]$ with $s[v]$ and convert $\sigma[1, u-1]$ to $t(u-1, v-1)$ with the minimum number of operations (i.e., $m(u-1, v-1)$)

From the above, we can obtain the recursive formula for computing the edit distance by dynamic programming as follows:

$$\begin{aligned} m(u, v) = \min\{&m(u-1, v-1) + \delta(\sigma[u], s[v]), \\ &m(u-1, v) + 1, m(u, v-1) + 1\}, \end{aligned} \tag{3.2}$$

		string											
		J	a	c	k	s	o	n	v	i	l	l	e
	0	0	0	0	0	0	0	0	0	0	0	0	0
J	1	0	1	1	1	1	1	1	1	1	1	1	1
a	2	1	0	1	2	2	2	2	2	2	2	2	2
c	3	2	1	0	1	2	3	3	3	3	3	3	3
k	4	3	2	1	0	1	2	3	4	4	4	4	4
i	5	4	3	2	1	1	2	3	4	4	5	5	5
e	6	5	4	3	2	2	2	3	4	5	5	6	5

query

Figure 3.5: Computation of the substring edit distance

where $\delta(c_1, c_2)$=0 if c_1=c_2 and $\delta(c_1, c_2)$=1 otherwise. Initially, we set $m(u, 0) = u$ for every u with $0 \leq u \leq |\sigma|$ and set $m(0, v) = 0$ for every v with $0 \leq v \leq |s|$ since the minimum number of operations to transform an empty string to a substring of s is 0.

Finally, considering all possible end positions of substrings in s, the substring edit distance between σ and s, which is referred to as $d_{sub}(s, \sigma)$, is obtained by

$$d_{sub}(s, \sigma) = \min_{1 \leq v \leq |s|} \{m(|\sigma|, v)\}. \tag{3.3}$$

Example 3.2.3 Consider a query string σ='Jackie' and a string s='Jacksonville'. In Figure 3.4, we present $m(u, v)$ values computed by the dynamic programming algorithm. Initially, $m(u, 0)$s with $0 \leq u \leq 6$ are set to u and $c(0, v)$s with $0 \leq v \leq 12$ are set to 0. Then, we compute $m(u, v)$s in the same fashion as we calculate $c(u, v)$s when computing the edit distance between σ and s. For example, to compute $m(5, 9)$ which is the minimum edit distance between two substrings 'Jacki' and a substring which ends at the 9-th position of 'Jacksonville', since the last letters are the same with 'i', $\sigma[1, 5]$='Jacki' can be matched to either the substring $s[1, 9]$='Jacksonvi' with 4 insertions or $s[5, 9]$='sonvi' with 4 substitutions. Finally, we get the substring edit distance which is $m(6, 4)$=2. ∎

3.3 Size Estimation of Similarity Query Result

Estimating the result sizes of similarity queries precisely is very important when we find optimal query plans for processing the similarity queries. To estimate the result sizes of exact substring matching queries, Jagadish et al. proposed MO method in [35] based on the Markov assumption and the concept of maximally overlapping substrings. Chen et al.[101] used a suffix tree and MinHash signatures [19] to estimate the selectivity of Boolean predicates for substring matching.

To estimate the result sizes of approximate substring matching queries, Jin and Li[36] proposed SEPIA which builds clusters of similar strings and maintains histograms that store the distribution information of strings with augmented edit distance information. The extension of q-grams with wildcards for estimating selectivity of string matching with edit distance was also proposed by Lee, Ng and Shim in [47].

To estimate the query result sizes of both exact and approximate substring matching, the MinHash technique proposed in [101] was shown to be the most precise[101, 47]. Thus, we utilize the MinHash technique to estimate the query result size in our works.

MinHash technique: Min-wise independent permutation, which is called *MinHash*, is a well-known Monte-Carlo technique that estimates Jaccard coefficient between sets A and B, which is represented by $|A \cap B|/|A \cup B|$. Based on a probabilistic analysis, Cohen et al. [19] proposed unbiased estimators to estimate the size of a set by repeatedly assigning random ranks to the universe and keeping the minimal rank of a set. Each assignment of random ranks to the universe is a random permutation.

	r_1	r_2	r_3	r_4	r_5
π_1	1	5	3	2	4
π_2	3	2	1	4	5
π_3	4	3	5	1	2
π_4	5	4	2	3	1

(a) Permutations

Set	Signature
A=$\{r_1,r_3,r_5\}$	<1,1,2,1>
B=$\{r_4,r_5\}$	<2,4,1,1>
C=$\{r_1,r_5\}$	<1,3,2,1>

(b) Signatures of set A, B, C

Figure 3.6: An example of set resemblance

The minimum values from the permuted ranks of a set kept for each permutation are called a *MinHash signature* and can also be used to estimate the Jaccard coefficient.

We now present the formal definitions. Consider a set of random permutations $\Pi = \{\pi_1, \cdots, \pi_{|\Pi|}\}$ on a universe $U \equiv \{r_1, \cdots, r_{|U|}\}$. For a set $A \subset U$, let $\min(\pi_i(A))$ denote $\min\{\pi_i(x)|x \in A\}$. Π is called *min-wise independent* if for any subset $A \subset U$ and any $x \in A$ for all $\pi_i \in \Pi$, we have $Pr(\min(\pi_i(A)) = \pi_i(x)) = 1/|A|$. The signature vector of A is constructed as: $sig_A = \langle \min(\pi_1(A)), \cdots, \min(\pi_{|\Pi|}(A)) \rangle$ and similarly for sig_B. The i-th entry of vector sig_A is denoted as $sig_A[i]$ or $\min(\pi_i(A))$. By matching the signatures sig_A and sig_B per permutation, the Jaccard coefficient between A and B can be approximated as: $|\{i|sig_A[i] = sig_B[i]\}|/|\Pi|$.

For k sets, $A_1, \ldots, A_k$, we can estimate the Jaccard coefficient ρ of the k sets, which is $|A_1 \cap \ldots \cap A_k|/|A_1 \cup \ldots \cup A_k|$, as follows:

$$\hat{\rho} = \frac{|\{i|sig_{A_1}[i] = \cdots = sig_{A_k}[i], 1 \leq i \leq |\Pi|\}|}{|\Pi|} \tag{3.4}$$

To estimate the size of $A_1 \cap \cdots \cap A_k$, we need to estimate the size of $A_1 \cup \cdots \cup A_k$ as follows. Let $A = A_1 \cup \cdots \cup A_k$ and $A_j \in \{A_1, \cdots, A_k\}$. Let the Jaccard coefficient

of A and A_j be γ=$|A \cap A_j|/|A \cup A_j|$. Since $A \cap A_j = A_j$ and $A \cup A_j = A$, we get:

$$|A| = |A_1 \cup \cdots \cup A_k| = \frac{|A_j|}{\gamma}. \tag{3.5}$$

A_j in Equation (3.5) can be any set $A_j \in \{A_1, \cdots, A_k\}$, but the one whose size is the biggest is known to give generally the best performance in [101]. To estimate γ, the signature of union, sig_A, can be constructed with the individual signatures as follows:

$$sig_A[i] = \min(sig_{A_1}[i], \cdots, sig_{A_k}[i]). \tag{3.6}$$

Thus, we can estimate γ as $\hat{\gamma}$ = $|\{i|sig_A[i] = sig_{A_j}[i], 1 \leq i \leq |\Pi|\}|$ / $|\Pi|$ and the size of $|A_1 \cup \ldots \cup A_k|$ as $|A_j|/\hat{\gamma}$. Finally, $|A_1 \cap \ldots \cap A_k|$ can be estimated as $\hat{\rho} \cdot |A_j|/\hat{\gamma}$.

Example 3.3.1 Consider the three sets $A = \{r_1, r_3, r_5\}$, $B = \{r_4, r_5\}$ and $C = \{r_1, r_5\}$ where the entire universe of record IDs is $U = \{r_1, \cdots, r_5\}$. Assume four random permutations $\pi_1, \cdots, \pi_4$ as in Figure 3.6(a), where r_i represents the rank of the record ID i. Based on Equation (3.6), the MinHash signature $sig_{A \cup B \cup C}$ is $\langle 1, 1, 1, 1 \rangle$. Now applying Equation (3.5) to estimate the resemblance between $A \cup B \cup C$ and A gives $\hat{\gamma} = 3/4$. Then, we can estimate the size of $|A \cup B \cup C|$ which is $|A|/\hat{\gamma} = 4$. This turns out to be exact, as $A \cup B \cup C = \{r_1, r_3, r_4, r_5\}$. We next estimate ρ with Equation (3.4) as $1/4$ because the signature values match only for π_4. The size of $|A \cap B \cap C|$ can be estimated as $\hat{\rho} \cdot |A|/\hat{\gamma} = 1$, which is the exact size of $A \cap B \cap C = \{r_5\}$. ∎

It is reported in [14, 101] that in practice, $|\Pi|$ does not need to be big for estimating the Jaccard coefficient. Chen et al. used 50 permutations when $|U|$ is about 2^{17} in [101]. Furthermore, Broder et al. in [14] proved that the family of permutations of the form $\pi(x) = ax + b \mod |U|$ with $a \neq 0$ performs well in practical.

Chapter 4

Exact Substring Matching

Consider a relational table schema *papers(record_id, title)* where *record_id* and *title* columns contain the record ID and title of each paper respectively. We may be interested in finding all papers whose titles contain a string 'text'. We can represent the substring matching query as `select * from papers where title like '%text%'`. Due to the popularity of such substring matching queries, there have been many studies on designing efficient indexes to support the LIKE clauses in SQL. (See [61] for a survey of data structures and algorithms for string matching queries.)

Let σ be a string. Any substring of length $q(> 0)$ in σ is called a q-gram. A positional q-gram comprises a q-gram and its position in σ. For example, the 3-grams of the string 'string' are {'str', 'tri', 'rin', 'ing'} and the positional 3-grams of 'string' are {('str',1), ('tri',2), ('rin',3), ('ing',4)}. An inverted index of positional q-grams over a column of a table consists of all q-grams appearing in the column. Assume that the *papers* table has the records of {(r_1, 'string'), (r_2, 'substring')}. The inverted index with various positional q-grams for the papers relation is shown in

Figure 4.1(a) where there are seven 3-grams, two 4-grams and a 5-gram. The entry for 3-gram 'str' has the list of {(r_1,1), (r_2,4)} which represents that the substring 'str' appears in the first position of the first record and the fourth position in the second record. Note that the posting list for each q-gram is presented in increasing order of record ID first and position next.

To process the example query that finds all papers whose titles contain 'string', we first generate all possible positional 3-grams of 'string' which are {('str',1), ('tri',2), ('rin',3), ('ing',4)} and perform the intersection of 4 posting lists of these 3-grams. When we perform the intersection between the posting lists of two positional q-grams of the query string, for each pair of postings in the two posting lists respectively, not only the record IDs should be matched, but the difference in positions between the postings should also be the same as the difference in the positions of the two positional q-grams in the query string. Thus, the result of intersection becomes {(r_1,1),(r_2,4)} which coincides with the fact that both records in the table contain 'string' starting from the first and fourth positions respectively. Since every posting list is sorted, we can simply perform a merge join style intersection. Thus, the I/O cost of the intersection can be estimated as the number of read pages and it is estimated as the sum of the numbers of the pages of all intersected posting lists.

Instead of using q-grams with a fixed value of q, it has been shown previously [53, 96] that it is better to use a set of variable-length q-grams with $q_{min} \leq q \leq q_{max}$, called VGRAM, to improve the performance of approximate string matching with the edit distance as a similarity measure. Even though VGRAM was not proposed for substring matching, we can still utilize it for substring matching.

In VGRAM, from a list of strings called a *gram dictionary* is built, which maintains

gram	Posting list
sub	(r_2,1)
ubs	(r_2,2)
bst	(r_2,3)
str	(r_1,1), (r_2,4)
tri	(r_1,2), (r_2,5)
rin	(r_1,3), (r_2,6)
ing	(r_1,4), (r_2,7)
ubst	(r_2,2)
bstr	(r_2,3)
bstri	(r_2,3)

(a) An inverted index

gram	Number of pages
sub	21
ubs	56
bst	44
str	202
tri	33
rin	13
ing	180
ubst	9
bstr	32
bstri	30

(b) Posting list sizes

Figure 4.1: An example of inverted index using variable-length grams

a subset of q-grams with the values of q ranging from q_{min} to q_{max}. Given a gram dictionary, from a list of strings to be indexed, all q_{min}-grams and variable-length grams appearing in the gram dictionary are indexed. For instance, Figure 4.1(a) is an inverted index of variable-length grams with $q_{min} = 3$ and $q_{max} = 5$ for the table {(r_1, 'string'), (r_2, 'substring')}, where the gram dictionary has the grams in {'sub', 'ubs', 'ubst', 'bst', 'bstr', 'bstri', 'str', 'tri', 'rin', 'ing'}. Then, with a query string, the gram dictionary is used to locate the variable-length grams to be used to perform the intersection of posting lists for substring matching. Indexing with variable-length grams may increase the sizes of the inverted indexes, but we can reduce the query processing time since the longer variable-length grams may have higher selectivities.

For a substring matching query that retrieves all records containing 'substring', we have several evaluation methods to answer the query with an inverted index

of positional variable-length grams. The first approach is intersecting the posting lists of all 3-grams {'sub', 'ubs', 'bst', 'str', 'tri', 'rin', 'ing'} appearing in the query string. For simplicity, instead of performing a 2-way intersection with two posting lists repeatedly, we assume that all posting lists are intersected together at once in order to save the I/O cost by not producing temporary results of 2-way intersections over and over again. We will consider I/O cost which is the number of page accesses to a disk only for the estimation of query evaluation cost. Assume that the number of records containing 'substring' is 10 and each record occupies exactly 1 page in the disk. Then, when the number of pages occupied by the posting lists are given as in Figure 4.1(b), the total I/O cost which is the sum of the number of read pages of all posting lists and the number of retrieved pages of the result records is 559(=549+10) where 549 is the summation of I/O costs for all 3-grams.

Instead of using all q-grams in the query, however, we may use any *covering variable-length gram set*, in which the character of every position in the query string is covered by at least a variable-length gram. For example, the variable-length gram sets such as {'sub', 'str', 'ing'}, {'sub', 'bstri', 'ing'} and {'sub', 'ubst', 'tri', 'ing'} are covering variable-length gram sets, for a string 'substring'. Since we need at least three variable-length grams to cover the query, {'sub', 'str', 'ing'} and {'sub', 'bstri', 'ing'} are called *minimal covering variable-length gram set*s. For the sets {'sub', 'str', 'ing'} and {'sub', 'bstri', 'ing'}, the I/O costs are 413(=21+202+180+10) and 241(=21+30+180+10) respectively. Note that the I/O cost of using the minimum covering set is much smaller than the cost of joining the posting lists of all 3-grams in the query. Furthermore, the I/O cost for the set {'sub', 'ubst', 'rin', 'ing'}, is 233(=21+9+13+180+10) which is smaller than those of all variable-length gram

sets considered previously with smaller number of grams.

Until now, the intersection result contains exactly all of the matching records since we considered covering variable-length gram sets. We can even use the posting lists of a non-covering variable-length gram set to answer substring matching queries. In this case, we need a filtering step for checking whether each record of the id in the intersection result contains the substring, because there are non-checked characters by intersecting the posting lists of a non-covering variable-length gram set.

Assume the number of records containing 'ubstri' is 60 and let us perform the intersection with posting lists of 'ubst' and 'bstri'. Then the I/O cost of the intersection is 39(=9+30). Since we performed the intersection of the posting lists of 'ubst' and 'bstri', we have to retrieve 60 records in the intersection result that is larger than 10, but the total I/O cost, which is 99(=39+60), is much smaller than the I/O costs of the previous methods using covering variable-length gram sets of the query. However, if we assume the number of records containing 'ubstri' is 200 instead, the total I/O cost becomes 239(=39+200) which is bigger than that of using the covering variable-length gram sets. This example justifies the importance of choosing the optimal variable-length gram set judiciously.

To the best of our knowledge, existing methods for answering substring matching queries using the inverted indexes with positional fixed or variable length grams perform an intersection of the posting lists of covering gram sets[3, 53, 65, 40, 96, 105]. However, as we illustrated previously, there are many alternative methods of using subsets of not only fixed-length grams but also variable-length grams to process a given substring query and we have to judiciously select the best one.

4.1 Query Processing Cost of Substring Matching

We first dicuss how to estimate I/O cost of processing substring matching queries with an inverted index utilizing vq-grams, and then we present our problem formulation of finding an optimal plan of substring matching with the minimal I/O cost.

To process a query with substring matching, we have to bring each record in the intersection result into main memory, which incurs I/O cost of reading pages. After bringing each record, we check the other selection conditions in the query, if any, and output the requested column values of the records satisfying all selection conditions. Since CPU cost is very cheap compared to I/O cost, when we estimate the cost of query processing, only I/O cost is generally considered.

Cost formula: Let Q denote a set of positional vq-grams $\{(g_1, p_1), (g_2, p_2), \ldots, (g_{|Q|}, p_{|Q|})\}$ for a given query string σ. Without loss of generality, we assume that (g_i, p_i) is ordered in increasing order of the position p_i. Given an inverted index using vq-grams, let L_i denote the posting list of a vq-gram g_i and $|L_i|$ denote the number of entries (i.e. postings) of posting list L_i. We also let ℓ_i denote the number of pages occupied to store the posting list L_i in disk. We can store the posting list of every possible vq-gram in an extendible hash or a B+ tree index in disk [77]. When we store the posting list in the B+ tree, ℓ_i is the number of leaf nodes in disk having the posting list L_i. For a given positional vq-gram set Q for processing a query, the I/O cost of reading posting lists is the sum of every ℓ_i. Furthermore, the I/O cost for bringing the records in the intersection result to check the rest of the selection conditions should be considered.

When we compute the I/O cost of reading the records in the intersection result with random references, we must consider the buffering effect. Since every page of the posting lists is brought to memory only once with each query to process intersection and the sizes of posting lists are generally large, we will simply assume that the traversal of the inverted index does not have any page hit in our I/O cost formula. Thus, we will consider buffering effect only for reading the record in the intersection result. Let $f_B(n)$ be the number of page accesses (i.e. page faults) to disk with B buffer pages in main memory for bringing n pages in disk. Then, assuming the height (i.e., depth) with the B+ tree or extendible hashing is h, the I/O cost of the query processing with a positional vq-gram set Q is defined as

$$Cost(Q) = \sum_{(g_i,p_i)\in Q} (h+\ell_i\text{-}1) + f_B(\left|\Pi_{rid}\left(\hat{\cap}_{(g_i,p_i)\in Q} L_i\right)\right|). \tag{4.1}$$

When we use the extendible hashing, if we keep the directory in main memory to access the posting lists, we have $h = 1$ for the above cost formula.

When there are $|R|$ record IDs and B pages of buffer are given, the expected number of page faults while we access n records with random references is given in [27]:

$$f_B(n) = \begin{cases} |R| \cdot [1 - (1 - 1/|R|)^n], & \text{if n} < k_0, \\ B + (n - k_0)(1 - B/|R|), & \text{if n} \geq k_0 \end{cases} \tag{4.2}$$

where k_0=$\ln(1\text{-}B/|R|)/\ln(1\text{-}1/|R|)$ which is the expected number of records to access until the B buffers get full.

Now we are ready to give a formal definition of the problem of the optimal processing of substring matching with the minimal I/O cost.

Problem definition: Given a set of documents D and a query string σ, the problem

of optimal substring matching using an inverted vq-gram index is to find all substrings in D containing σ with the minimal I/O cost, which is measured by Equation (4.1).

In the remainder of this chapter, we shall investigate how we can minimize the I/O cost when we perform substring matching using an inverted vq-gram index.

4.2 Traditional Substring Matching Algorithms

The most simple method for query processing using an inverted vq-gram index is to perform the intersection of all posting lists of vq-grams in a query string and return the distinct records. We refer to this simple algorithm as *MAX-COVER*. We next describe the other traditional substring matching algorithms *MERGE-OPT* and *OPT-MINC*.

MERGE-OPT: Since performing the intersections of the posting lists of all the vq-grams in a query string requires too many computations, the algorithm *MergeOpt* in [73], which was proposed for set similarity join using an inverted index, can be extended to reduce the number of intersection operations for substring matching with the inverted vq-gram index. The intuition is, for each posting in a posting list, to directly search the exact matching postings from all the other posting lists using an index such as B+tree or extendible hash index. The algorithm first retrieves the shortest posting list among the vq-grams of query string σ. Let g_i be the vq-gram with shortest posting list starting at the p_i-th position of σ. For each post (r,p) in the posting list of g_i, we examine the posting list of every other vq-gram g_j, which starts at the p_j-th position in σ, to check whether there is any posting (r,p') where $p_i - p_j$

Function MERGE-OPT(σ, D)
σ: a query string
D: vq-gram dictionary
begin
1. Q = vq-gram set which covers σ;
2. (g_i, p_i) = the positional vq-gram in Q with the smallest posting list;
3. L = the posting list of g_i;
4. $u = 0$;
5. **for** v=1 to $|L|$ **do** {
6. (r,p) = $L[v]$;
7. **for each** positional vq-gram (g_j, p_j) and $i \neq j$ **do**
8. **if** the posting list of g_j has (r,$p + p_j - p_i$) **then**
9. $u = u + 1$;
10. $L[u] = L[v]$;
11. exit for;
12. }
13. }
14. **return** $\Pi_{rid}(L[1..u])$;
end

Figure 4.2: MERGE-OPT

$= p - p'$. When we use a B+ tree, we can optimize the I/O cost by searching the postings in the B+ tree index. When the inverted indexes are built with extendible hashing and each posting list is stored sequentially, we can search the postings with binary searches.

The algorithm *MERGE-OPT* is shown in Figure 4.2. First, we generate the vq-gram set Q from the query string σ and select the vq-gram whose posting list size is smallest. Let us assume that the vq-gram g_i at p_i-th position in σ has the smallest

posting list. For each posting (r,p) in the posting list, we search for the posting (r,$p + p_j - p_i$) in the posting list of vq-gram g_j, which is located at the p_j-th position in σ, because the difference of positions between p and $p+p_j-p_i$ is the same as that of g_i and g_j. If there is any posting list which does not have any matching posting, we do not include the posting (r,p) in the result list. After repeating this step for all postings in the smallest posting list, we finally obtain the result of the substring matching query.

MERGE-OPT may have smaller computation cost than *MAX-COVER*. However, they still utilize all posting lists of the vq-grams in a query string. Thus, we will introduce an algorithm that reduces the I/O cost by reading the posting lists of covering vq-grams only.

OPT-MINC: If we use fixed length grams, since we can use any covering positional q-gram set to obtain the same result with the intersection of the posting lists of all q-grams, the dynamic programming algorithm for finding a minimal covering q-gram set with the minimal I/O cost of reading posting lists is proposed in [65]. We will refer to this algorithm as *OPT-MINC*.

For a query string σ whose length n is a multiple of q, there is only a single minimal covering q-gram set, $\{\sigma[1,q]$,$\sigma[q+1,2q]$,$\cdots$,$\sigma[n-q+1,n]\}$, with n/q q-grams and the sum of the I/O costs of reading posting lists can be represented as $\sum_{i=0}^{n/q-1} c_p(\sigma[iq{+}1,(i{+}1)q])$ where $c_p(\sigma[iq{+}1,(i{+}1)q])$ denotes the cost of reading the posting list of the q-gram $\sigma[iq{+}1,(i{+}1)q]$. However, when the query string length n is not a multiple of q, if n is shorter than $\lceil n/q \rceil * q$ by $k{=}\lceil n/q \rceil * q - n$ characters, the right boundary of the q-gram $\sigma[j{-}q{+}1,j]$ on the left side of the q-gram $\sigma[n-q+1,n]$ is located at the position $n-q+x$ with $0 \leq x \leq k$. For instance, when

the query string σ has 8 characters, we have 2 minimal covering 3-gram sets such as $\{\sigma[1,3]$,$\sigma[3,5]$,$\sigma[6,8]\}$ and $\{\sigma[1,3]$,$\sigma[4,6]$,$\sigma[6,8]\}$.

Let $m[1,j]$ be the optimal cost of performing an intersection of posting lists in a covering q-gram set of $\sigma[1,j]$. Note that the sizes of the intersection results are all the same for all covering q-gram sets, so we can ignore the cost of bringing the actual records into main memory in the filtering step for cost comparisons. Then, we obtain the recursive formula below:

$$m[1,i] = \min_{0 \le x \le k} \{m[1, i-q+x] + c_p(\sigma[i-q+1, i])\}, \tag{4.3}$$

where $k = \lceil i/q \rceil * q - i$ and $m[1,q] = c_p(\sigma[1,q])$.

The optimal cost $m[1,j]$ is the minimum one among the sums of the costs of processing the substrings $\sigma[1, i-q+x]$ optimally and the cost of accessing the q-gram $c_p(\sigma[i-q+1,i])$ together for $0 \le x \le k$. It compares x number of overlapping substrings such as $\sigma[1,aq]$,$\sigma[1,aq-1]$,$\cdots$ and $\sigma[1,aq-x]$ with the last q-gram. For example, with the query string σ of 8 characters and $q = 3$, $m[1,n]$ is computed by selecting the minimum between $m[1,5]$+$c_p(\sigma[6,8])$ and $m[1,6]$+$c_p(\sigma[6,8])$.

4.3 Our Substring Matching Algorithms

In this section, we will present our algorithms for optimal execution of substring matching queries with inverted vq-gram indexes. Since the search space of finding the optimal vq-gram set among all possible vq-gram sets is exponential with the size of the vq-gram set, we next present the effective pruning strategy to reduce the search space. Using the pruning rule, we present the extended traditional algorithms. Then we extend the optimal algorithm of finding the optimal and propose

optimal algorithms that find the optimal Finally, we present the optimal and approximate algorithms based on the cost estimation using the pruning strategy for substring matching queries.

Note that to find the best vq-gram set which minimizes the I/O cost, we need to estimate the size of the intersection result between the posting lists of vq-grams. In order to select the optimal vq-gram set in our algorithms, we assume that a MinHash signature is stored for the posting list of each vq-gram and we will use the signature for estimating the sizes of the intersections of posting lists as presented in Section 3.3.

4.3.1 Efficient Pruning of Search Space

Finding the optimal vq-gram set that minimizes the cost of Equation (4.1) among all possible vq-gram subsets requires an exponential number of cost estimations. Formally, given a vq-gram dictionary D with q_{min} and q_{max}, let D_σ be the set of all vq-grams of σ in D. For a given query σ of length n, the size of D_σ is at most $g(n)$=$(n+1-(q_{max}+q_{min})/2)\cdot(q_{max}-q_{min}+1)$ and thus we have $O(2^{g(n)})$ number of possible sets of variable-length grams to answer the query. Since when we use q-grams with a fixed size q only, the number of q-grams for σ is at most $n-q+1$ and we have $O(2^{n-q+1})$ number of possible sets of q-grams, the search space of using vq-grams is much larger than that of fixed q-grams only. Thus, to select the set of vq-grams with the minimal I/O cost is much more difficult than to select the optimal set of q-grams with a fixed size q. Fortunately, the following observation allows for the reduction of the large search space when we find the set of vq-grams with the optimal I/O cost among all possible sets of vq-grams for a query string.

Observation 4.3.1 *Given a query string σ and a vq-gram dictionary D, let Q be an optimal vq-gram set of σ. Then, for a vq-gram g in Q, if g is a substring of a vq-gram g' in D_σ with $g' \neq g$, the I/O cost of a vq-gram set Q', which is obtained by replacing g with g' in Q, is at most the cost of Q.*

Proof: Let L_g and L'_g be the posting lists of g and g' respectively. ℓ_g and ℓ'_g are the number of pages storing posting lists L_g and L'_g respectively. Since g is a substring of g', every record with g' also contains g and L'_g equal to or is a subset of L_g. Thus, $A \hat{\cap} L_g \supseteq A \hat{\cap} L'_g$ holds by a well-known set theory where A is any posting list or intermediate intersection result. Therefore, we have the inequality $|A \hat{\cap} L_g| \geq |A \hat{\cap} L'_g|$. Since $L_g \supseteq L'_g$, we have the inequality of $\ell_g \geq \ell'_g$ as well.

Then we arrive at the following inequality:

$$\begin{aligned} Cost(Q) &= \sum_{(g_i,p_i)\in Q/\{(g,p)\}} (h+\ell_i\text{-}1) + (h+\ell_g\text{-}1) \\ &\quad + f_B(\left|\Pi_{rid}\left(\left(\hat{\cap}_{(g_i,p_i)\in Q/\{(g,p)\}} L_i\right) \hat{\cap} L_g\right)\right|) \\ &\geq \sum_{(g_i,p_i)\in Q/\{(g,p)\}} (h+\ell_i\text{-}1) + (h+\ell'_g\text{-}1) \\ &\quad + f_B(\left|\Pi_{rid}\left(\left(\hat{\cap}_{(g_i,p_i)\in Q/\{(g,p)\}} L_i\right) \hat{\cap} L'_g\right)\right|) \\ &= Cost(Q'), \end{aligned}$$

where the operator '/' denotes the set difference operator.

Since $f_B(n)$ is a monotonically increasing function, in other words, the inequality $f_B(n) \geq f_B(n')$ holds for $n \geq n'$, we obtain the inequality $Cost(Q) \geq Cost(Q')$ for g and g' when g is a substring g'. Thus, Q' is also an optimal vq-gram set.∎

Let D^R_σ be a set of vq-grams from D_σ which has every vq-gram that is not a substring of any other vq-gram in D_σ. Then, according to Observation 4.3.1, there

is always an optimal vq-gram set in which every vq-gram is in D_σ^R. Thus, we obtain the following lemma.

Lemma 4.3.2 *Given a query string σ and a vq-gram dictionary D, we can find an optimal vq-gram set Q by exploring the subsets of D_σ^R only. That is, Q is a subset of D_σ^R.*

We can also obtain the following corollary.

Corollary 4.3.3 *Given a query string σ and a vq-gram dictionary D, let Q be the optimal vq-gram set obtained by exploring D_σ^R. Then, for every position of σ, there is at most one vq-gram in Q which starts at the position and there is at most one vq-gram in Q that ends at the position.*

Proof: If two vq-grams in Q start at the same position or end at the same position, one of the two vq-grams must be a substring of the other vq-gram. This contradicts Lemma 4.3.2 that Q is a subset of D_σ^R.∎

Generation of D_σ^R: When using a vq-gram dictionary D to generate D_σ^R for a string σ, the length of vq-grams to generate from σ varies depending on the string σ and the vq-grams in D. Intuitively, while we increase the current position from 1 to $|\sigma| - q_{min} + 1$, we generate a vq-gram that is the longest substring (starting from the current position) that matches a vq-gram in D and also is not contained by any of the longest substrings generated previously.

Formally, we decompose string σ to D_σ^R using the algorithm *MAX-VQGRAM* in Figure 4.3, which is borrowed from the decomposition algorithm in [53]. Initially, V is set to an empty set and we start by setting the current position to 1. In each step, we find the longest substring of σ in the vq-gram dictionary D by searching from the current position. The function *LONGEST-PREFIX*(D, s) returns the longest

Function MAX-VQGRAM(D, σ, q_{min}, q_{max}, S, E)
begin
1. $V = \{\}$, $start = 1$, $prev = null$;
2. **while** $start \leq |\sigma|$-q_{min}+1 **do** {
3. g = LONGEST-PREFIX(D, $\sigma[start, |\sigma|]$);
4. **if** g is $null$ **then** {
5. $V = V \cup \{(\sigma[start, start+q_{min}\text{-}1], start)\}$;
6. $S[start] = q_{min}$;
7. $E[start+q_{min}\text{-}1] = q_{min}$;
8. } **else if** ($prev$ is $null$) or ($g \not\subseteq prev$) **then** {
9. $V = V \cup \{(g, start)\}$;
10. $S[start] = g.length$;
11. $E[start+g.length\text{-}1] = g.length$;
12. $prev = g$;
13. }
14. $start = start+1$;
15. }
16. return V;
end

Figure 4.3: MAX-VQGRAM

prefix of s that appears in D. Next we check whether the returned vq-gram is a substring of the previous vq-gram, because the returned vq-gram should not be a substring of any other vq-gram in D_σ^R. If the returned vq-gram is not a substring of any other vq-gram, we put it into the set V and we move the current position to the right by one character.

Example 4.3.4 Consider the string 'substring', and a vq-gram dictionary with $q_{min} = 3$ and $q_{max} = 5$ which is shown in Figure 3.2. With setting $start = 1$ and $V = \{\}$, we start from the first position of 'substring'. Since the longest prefix starting from

position 1 in D is 'sub', we insert it into V. Then the algorithm sets $start$ to 2. Next we insert 'ubs', which is the longest prefix appearing in D at the position of 2, into V and update $start$ to 3. Similarly, we also insert 'bst' and 'strin' into V. At the next step where $start$ is 5, the longest prefix starting from position 5 is 'tri', but it is not inserted into V because 'tri' is a substring of 'strin' that is already included in V. When $start$ becomes $|\sigma| - q_{min} + 1$=7, we get V={'sub', 'ubs', 'bst', 'strin', 'ing'} for D_σ^R. ∎

Since *MAX-VQGRAM* executes *LONGEST-PREFIX* at each position of σ and each invocation of *LONGEST-PREFIX* takes up to q_{max} comparisons for searching a trie, the time complexity is O(nq_{max}).

Reduction of search space for traditional algorithms: The simplest method for query processing with inverted vq-gram indexes is to perform the intersection of the posting lists of all vq-grams in D_σ^R only instead of all vq-grams of σ. Thus, we can extend the algorithms *MAX-COVER* and *MERGE-OPT* in Section 4.2 to use D_σ^R only which is generated by *MAX-VQGRAM*.

We first present the algorithms using covering vq-gram sets and next propose the algorithms for both covering and non-covering vq-gram sets.

4.3.2 Optimal Algorithms for Covering vq-gram Sets

The dynamic programming algorithm *OPT-MINC* in Section 4.2, which finds the optimal covering vq-gram set among the minimal covering sets, is designed for q-grams of a fixed length. Thus, we extend *OPT-MINC* to explore the covering vq-gram sets with the minimum number of vq-grams. We call the extended algorithm *OPT-VMINC*.

OPT-VMINC: Let $n[1, j]$ be the minimal size of covering vq-gram set of $\sigma[1, j]$ and γ_j^* be the length of the vq-gram in D_σ^R that ends at the position of j. Thus, $\sigma[j\text{-}\gamma_j^*+1, j]$

is the vq-gram that ends at j-th position in D_σ^R. Consider the optimal covering vq-gram set $\theta[1,j]$ that covers the range of $[1,j]$ minimally. Then the rightmost vq-gram in $\theta[1,j]$ is $\sigma[j\text{-}\gamma_j^*\text{+}1,j]$ because only one vq-gram in D_σ^R ends at the j-th position by the Corollary 4.3.3. Since $\theta[1,j]$ should cover every position in $[1,j]$, not only $\theta[1,j]/\{\sigma[j\text{-}\gamma_j^*\text{+}1,j]\}$ should cover the positions from 1 to $j-\gamma_j^*$ but also the rightmost vq-gram in $\theta[1,j]/\{\sigma[j\text{-}\gamma_j^*\text{+}1,j]\}$ ends at a position in $[j\text{-}\gamma_j^*,j\text{-}1]$. Thus, we should consider the ending position of $\theta[1,j]/\{\sigma[j\text{-}\gamma_j^*\text{+}1,j]\}$ for every position in $[j\text{-}\gamma_j^*,j\text{-}1]$ and $n[1,j]$ can be computed by the following recursive formula:

$$n[1,j] = \begin{cases} \min\limits_{0\le x\le \gamma_j^*\text{-}1}\{n[1,j\text{-}\gamma_j^*\text{+}x]+1\} & \text{if } \sigma[j\text{-}\gamma_j^*\text{+}1,j]\in D_\sigma^R, \\ \infty & \text{otherwise,} \end{cases} \tag{4.4}$$

Let $m[1,j]$ be the cost of performing the intersection of posting lists in the optimal vq-gram set $\theta[1,j]$. $c_p(\sigma[j\text{-}\gamma\text{+}1,j])$ denotes the cost of reading the posting list of the vq-gram $\sigma[j\text{-}\gamma\text{+}1,j]$ and is defined as ∞ if D_σ^R does not contain the vq-gram. Since the rightmost vq-gram of $\theta[1,j]$ is $\sigma[j\text{-}\gamma_j^*\text{+}1,j]$, the optimal cost $m[1,j]$ is the minimum one among the sums of reading posting list cost for $\sigma[j\text{-}\gamma_j^*\text{+}1,j]$ and the cost of reading the posting lists of vq-grams which not only cover the substring $\sigma[1,j\text{-}\gamma_j^*]$ but also their rightmost vq-grams end in $[j\text{-}\gamma_j^*,j\text{-}1]$. Moreover, since $m[1,j]$ should be the optimal cost of reading posting lists of vq-grams that cover $\sigma[1,j]$ minimally, $n[1,j]$ must be $n[1,j\text{-}\gamma_j^*\text{+}x]$ + 1 with $0\le x\le\gamma_j^*\text{-}1$.

Then we obtain the recursive formulas for $m[1,j]$ below:

$$m[1,j]= \min_{\substack{0\le x\le\gamma_j^*\text{-}1\wedge \\ n[1,j\text{-}\gamma_j^*+x]+1=n[1,j]}} \{m[1,j\text{-}\gamma_j^*\text{+}x]+c_p(\sigma[j\text{-}\gamma_j^*\text{+}1,j])\} \tag{4.5}$$

Since *OPT-VMINC* considers only the minimal covering vq-gram sets among all

possible covering vq-gram sets in D_σ^R, the algorithm may not offer the optimal covering vq-gram set as the following example illustrates.

Example 4.3.5 Consider a query of 'festivals' with $q_{min} = 3$ and $q_{max} = 5$. Suppose that the cost $c_p(\sigma[i,j])$ of reading the posting list of the vq-gram $\sigma[i,j] \in D_\sigma^R$ is shown below:

'fest'	'sti'	'tiv'	'iva'	'val'	'als'
5	3	8	7	2	10

OPT-VMINC explores the minimal covering vq-gram sets such as {fest,tiv,als} and {fest,iva,als} only. Then, since c_p('fest') + c_p('tiv') + c_p('als')=23 and c_p('fest') + c_p('iva') + c_p('als')=22, *OPT-VMINC* chooses {fest,iva,als} between the two sets. However, note that the cost of non-minimal covering vq-gram set {fest,sti,val,als} is only 20 which is smaller than the cost of {fest,iva,als}. ∎

This example illustrates that the minimal covering vq-gram set with the cheapest cost may not be the covering vq-gram set with the optimal cost, so the selection of an optimal covering set should be done judiciously by exploring all covering vq-gram sets.

The reason why *OPT-VMINC* cannot explore non-minimal covering sets is because, in the recursive definition of $m[1,j]$ in Equation (4.5), it considers the vq-gram sets with $n[1,j\text{-}\gamma_j^*+x]+1=n[1,j]$ only to make the number of vq-grams minimal, while computing $m[1,j\text{-}\gamma_j^*+x]+c_p(\sigma[j\text{-}\gamma_j^*+1,j])$ for $m[1,j]$. Thus, for the above example, it does not consider $m[1,8]$ with c_p('als') while computing $m[1,9]$ because we have $n[1,8]+1 > n[1,9]$ where $n[1,8]+1=4$ and $n[1,9]=3$.

OPT-COVER: To overcome the above problem, we develop the dynamic programming algorithm *OPT-COVER* which considers the covering vq-gram sets with not only

minimal size but also non-minimal sizes by considering all vq-gram sets whose rightmost vq-gram ends in the range $[j\text{-}\gamma_j^*, j\text{-}1]$ while computing $m[1, j\text{-}\gamma_j^*+x] + c_p(\sigma[j\text{-}\gamma_j^*+1, j])$ for $m[1, j]$. Then, we also consider m[1,8] + c_p('als') to compute m[1,9] in the above example and we can find the optimal covering vq-gram set. The recursive definition of dynamic programming formulation for *OPT-COVER* is as follows:

$$m[1, i] = \min_{0 \le x \le \gamma_i^* - 1} \{m[1, i\text{-}\gamma_i^*+x] + c_p(\sigma[i\text{-}\gamma_i^*+1, i])\}. \quad (4.6)$$

Since *OPT-COVER* computes $m[1, i]$ with $q_{min} \le i \le n$ for a query string of length n and the computation cost for each $m[1, i]$ is $O(q_{max})$, its time complexity is $O(nq_{max})$.

4.3.3 Optimal Algorithms for All vq-gram Sets

OPT-COVER still considers the covering vq-gram sets only. However, since a non-covering vq-gram set may become an optimal vq-gram set, in order to find the optimal vq-gram set with the minimum cost of query processing, we can actually enumerate all possible subsets of D_σ^R only and choose the set with the minimum cost. Note that the size of the search space using all vq-gram sets is $O(2^{n\cdot(q_{max}-q_{min}+1)})$. However, since we have at most n-q_{min}+1 number of vq-grams in D_σ^R, we can enumerate $O(2^{n-q_{min}+1})$ subsets only if we use D_σ^R instead and we can reduce the size of the search space significantly. We will refer to this algorithm as **OPT-NAIVE** which takes $O(2^{n-q_{min}+1})$ time. Since the time complexity of *OPT-NAIVE* is exponential, we next propose an algorithm called *OPT-QSP* that prunes the vq-gram set that is guaranteed to be worse than the currently the best one in each step of searching.

Function OPT-QSP(σ, S, Q, τ, $OptQ$)
begin
1. **if** $|Q| \geq 2$ **then** $start$ = max ($Q[|Q|\text{-}2].end+1$, $Q[|Q|\text{-}1].start)+1$;
2. **else if** $|Q|$ is 1 **then** $start = Q[0].start+1$;
3. **else** $start = 1$;
4. **for** $k = start$ **to** $|\sigma|\text{-}q_{min}+1$ **do**
5. **if** $S[k] \neq null$ **then** {
6. $Q' = Q \cup \{\sigma[k, k+S[k]\text{-}1]\}$;
7. **if** $Cost(Q') < \tau$ **then**{
8. $OptQ = Q'$;
9. $\tau = Cost(Q')$;
10. }
11. **if** $readCost(Q') \leq \tau$ **then** OPT-QSP(σ, S, Q', τ, $OptQ$);
12. }
end

Figure 4.4: OPT-QSP

OPT-QSP: It starts from an empty vq-gram set and explores all possible vq-gram sets by adding another vq-gram one by one recursively. Let σ be a query string and Q be the current vq-gram set to be expanded. Let us also assume that $\alpha_{|Q|}$ and $\beta_{|Q|}$ denote the start and end positions of rightmost vq-gram of Q respectively. If Q is empty, $\alpha_{|Q|}$ is 0. We can explore all possible subsets of D^R_σ by adding vq-grams in D^R_σ to Q one by one, whose start positions are in the range $[\alpha_{|Q|}+1, |\sigma|]$, into the current set Q so that we do not examine previously explored vq-gram sets. Its time complexity is naturally exponential. However we can utilize the opportunities of pruning rules based on the following two observations.

Observation 4.3.6 *Consider a vq-gram set Q' that is generated by adding another vq-gram g into Q. If g is covered by at least two other vq-grams in Q', the intersection*

result of the posting lists of vq-grams in Q' and that of vq-grams in Q will be the same. Thus, the cost of retrieving the records in the intersection result is identical. Since Q' takes additional cost for reading the posting list of the vq-gram g than Q does, the total cost of Q' is larger than the total cost of Q. Therefore, we do not need to explore the expanded vq-gram set of Q' any more.

As a result of the above observation, we can expand Q by only adding a vq-gram where every vq-gram in Q is not covered by the added vq-gram and the other vq-grams in Q. Formally, let $\alpha_{|Q|-1}$ and $\beta_{|Q|-1}$ be the start and end positions of the second rightmost vq-gram in Q respectively. Assume that we expand Q by adding a vq-gram $g \in D_\sigma^R$ whose start position α_g is larger than $\alpha_{|Q|}$ (the end position β_g should be larger than $\beta_{|Q|}$ by the Lemma 4.3.2) but not larger than $\beta_{|Q|-1}+1$. Since the vq-grams in D_σ^R cannot be overlapped by the definition of D_σ^R, $\beta_{|Q|}$ is larger than $\beta_{|Q|-1}$. Then the rightmost vq-gram of Q, which is $\sigma[\alpha_{|Q|}, \beta_{|Q|}]$, will be covered by $\sigma[\alpha_{|Q|-1}, \beta_{|Q|-1}]$ and $\sigma[\alpha_g, \beta_g]$. Thus, we expand Q by adding a vq-gram whose start position is larger than both $\alpha_{|Q|}$ and $\beta_{|Q|-1}+1$.

Observation 4.3.7 *Since the cost of reading the posting lists of the vq-grams in Q always increases by adding a vq-gram into Q, if the cost of reading the posting lists of vq-grams in Q is at least the cost of the best current vq-gram set, the expanded vq-gram sets of Q cannot have smaller cost than the best cost so far.*

Due to the above observation, we do not consider the expansions of Q any more if the cost of reading posting lists of Q exceeds the best cost so far, since the cost of every expanded vq-gram set from Q' will definitely be larger than that of the current best one.

The pseudocode of *OPT-QSP*, which utilizes the above two pruning rules, is shown in Figure 4.4. Given the query string σ, the array Q maintains the vq-gram

Vq-gram set	Estimated size	Vq-gram set	Estimated size
{sub}	40	{sub, ubst}	6
{sub, ubst, bstr}	1	{sub, ubst, bstr, tri}	1
{ubst}	20	{ubst, bstr}	8
{ubst, bstr, tri}	2	{bstr}	60
{bstr, tri}	6	{tri}	70

Figure 4.5: An example of estimated intersection result sizes

set to be expanded and the vq-grams in Q are sorted in increasing order of start positions in σ. $Q[i].start$ and $Q[i].end$ are the start and end positions of the i-th vq-gram of Q respectively. S is an array of integers where $S[i]$ is the length of the vq-gram in D_σ^R that starts from the i-th position of σ. Note that each $S[i]$ has a unique value by Corollary 4.3.3. Let $Cost(Q)$ be the cost of processing a query with the vq-gram set Q as defined in Equation (4.1). The threshold τ is the best cost so far and $OptQ$ is the vq-gram set with the best cost so far. When *OPT-QSP* is invoked, we first set $start$, which is the start position of the vq-gram to be added, and is larger than not only $Q[|Q|].start$ but also $Q[|Q|-1].end$+1 due to Observation 4.3.6. Then, while we increase the start position of vq-gram to be added from $start$ to $|\sigma|$-q_{min}+1, we compare the reading posting list cost of the expanded vq-gram set Q' with τ. If the reading posting list cost of Q' is larger than τ, we do not expand Q' further by Observation 4.3.7. We start by calling *OPT-QSP*(σ, S, {}, ∞, $OptQ$) where S is obtained from calling *MAX-VQGRAM*(D, σ, q_{min}, q_{max}, S, E).

Example 4.3.8 Consider the vq-grams in Figure 4.1(b). Suppose that our query is 'substri' and D_σ^R is {(sub,1), (ubst,2), (bstr,3), (tri,5)}. For ease of presentation, instead of the cost formulation in Equation (4.1), we use $\sum_{(g_i,p_i)\in Q} \ell_i + |\hat{\cap}_{(g_i,p_i)\in Q} L_i|$ where L_i is the posting list of vq-gram g_i and ℓ_i is the number of pages in L_i.

$|\hat{\cap}_{(g_i,p_i)\in Q} L_i|$ of every possible vq-gram set is shown in Figure 4.5. Starting with Q={}, we first generate Q' by adding (sub,1) into Q and we call *OPT-QSP* recursively with Q'. In the recursive call, we produce Q' by appending (ubst,2) into Q and compute $Cost(Q')$= 21+9+6 = 36. In the next recursive call, when Q' is {(sub,1), (ubst,2), (bstr,3)}, $Cost(Q')$ becomes 63(=21+9+32+1) where $readCost(Q')$ is 62. Since $readCost(Q')$=62 is larger than $Cost(Q)$=36, we do not call *OPT-QSP* recursively with Q' any more. Similarly, we repeat the recursive calls and find the optimal vq-gram subset {(ubst,2)} with the cost 29. ∎

4.3.4 Approximate Algorithms

Since the time complexity of *OPT-QSP* is still exponential in the worst case, we will develop three approximate algorithms in this section.

APR-DPQ: The approximate algorithm *APR-DPQ* is based on dynamic programming. Let $\theta[1,i]$ be the vq-gram set of the minimal cost for processing the query $\sigma[1,i]$ and $m[1,i]$ be the query processing cost for the vq-gram set $\theta[1,i]$. When we compute the best cost $m[1,i]$, we consider the three cases for the best vq-gram set: the vq-gram set without $\sigma[i\text{-}\gamma_i^*+1,i]$, the vq-gram set with $\sigma[i\text{-}\gamma_i^*+1,i]$ alone and the vq-gram set containing $\sigma[i\text{-}\gamma_i^*+1,i]$. Thus, we select the minimum one among the costs of $\theta[1,x]$ for $q_{min} \le x \le j$-1, the vq-gram $\sigma[i\text{-}\gamma_i^*+1,i]$ alone and processing the query with $\theta[1,x]$ and $\sigma[i\text{-}\gamma_i^*+1,i]$ together for $q_{min} \le x \le j$-1. The recursive definition of dynamic programming formulation for *APR-DPQ* is as follows:

$$m[1,i] = \min_{q_{min}\le x\le i-1}\{\min(m[1,x], Cost(\{\sigma[i\text{-}\gamma_i^*+1,i]\}), \\ Cost(\theta[1,x]\cup\{\sigma[i\text{-}\gamma_i^*+1,i]\}))\}, \tag{4.7}$$

where $m[1,i]=\infty$ and $\theta[1,i]=\{\}$ with every $q_{min} \le i \le |\sigma|$ initially.

Note that *APR-DPQ* may not find the optimal vq-gram set with the minimum cost. Consider a vq-gram set S such that every vq-gram in the set is within the range of $[1, i\text{-}1]$ and assume that S does not coincide with $\theta[1, x]$ for $q_{min} \leq x \leq i\text{-}1$. When we compute $m[1, i]$, if the intersection result of the posting lists of vq-grams in S and the posting list of $\sigma[i\text{-}\gamma_i^*+1, i]$ have very small number of common records, the vq-gram set of $S \cup \{\sigma[i\text{-}\gamma_i^*+1, i]\}$ may have smaller cost than all of $m[1, x]$, $Cost(\{\sigma[i\text{-}\gamma_i^*+1, i]\})$ and $Cost(\theta[1, x] \cup \{\sigma[i\text{-}\gamma_i^*+1, i]\})$ for $q_{min} \leq x \leq j\text{-}1$. Thus, *APR-DPQ* may not find the optimal vq-gram set.

APR-DPQ requires $O(n)$ entries to compute the minimal cost of $\sigma[1, i]$ for $q_{min} \leq i \leq n$. Since we compute the cost of the vq-gram set $\theta[1, x] \cup \{\sigma[i\text{-}\gamma_i^*+1, i]\}$ with varying x from q_{min} to i-1, it takes $O(n)$ times to calculate each entry $m[1, i]$. Thus, the time complexity of *APR-DPQ* is $O(n^2)$ and the algorithm requires $O(n)$ space.

Example 4.3.9 Consider the vq-grams in Figure 4.1(b) and the estimated intersection result sizes of possible vq-gram sets are shown in Figure 4.5. Suppose that the query is 'substri' and D_σ^R is {(sub,1), (ubst,2), (bstr,3), (tri,5)}. When we compute $m[1, i]$ bottom-up, $\theta[1, 3]$ and $m[1, 3]$ are first set to {(sub,1)} and $Cost(\{$(sub,1)$\})$=21+40=61 respectively. For $m[1, 5]$, since the minimum value among $m[1, 3]$ = 61, $Cost(\{$(ubst,2)$\}$ = 29 and $Cost(\{$(sub,1), (ubst,2)$\})$ = 36 is 29, $m[1, 5]$ and $\theta[1, 5]$ are set to 29 and {(ubst,2)} respectively. Similarly, if we compute up to $m[1, 6]$, *APR-DPQ* returns {(ubst,2)} with the cost 29. ∎

APR-GRQ: This is a greedy approximation algorithm which stops adding a vq-gram when the vq-grams selected so far do not improve the performance of query processing despite adding the vq-gram. The algorithm of *APR-GRQ* is shown in Figure 4.6. *APR-GRQ* maintains a variable $OptQ$ which has the vq-gram set with the best cost so far. When the call *APR-GRQ*(D, σ, q_{min}, q_{max}) is executed, T is set to D_σ^R. Then we initialize $OptQ$ with an empty set and its cost with ∞. In each step of a do-while

Function APR-GRQ(D, σ, q_{min}, q_{max})
begin
1. T = MAX-VQGRAM(D, σ, q_{min}, q_{max}, S, E);
2. $OptQ.qset = \{\}$, $OptQ.cost = \infty$;
3. **while** (true) **do** {
4. $c_{min} = \infty$;
5. **for each** (g_i,p_i) $\in T$ **do** {
6. **if** $Cost(OptQ.qset \cup \{g_i\}) < c_{min}$ **then** {
7. $c_{min} = Cost(OptQ.qset \cup \{g_i\})$;
8. $g_{min} = g_i$;
9. }
10. }
11. **if** $c_{min} < OptQ.cost$ **then** {
12. $OptQ.qset = OptQ.qset \cup \{g_{min}\}$;
13. $OptQ.cost = c_{min}$;
14. $T = T$ - $\{g_{min}\}$;
15. } **else**
16. exit while loop;
17. }
18. **return** $OptQ$;
end

Figure 4.6: APR-GRQ

loop, we select the vq-gram with the best improvement by examining every vq-gram in T, add the vq-gram to $OptQ$ and remove the added vq-gram from T, if there is a vq-gram with improvement. We continue to the next iteration of the do-while loop if there is any improvement. Otherwise, we stop the loop. For selecting the i-th vq-gram in *APR-GRQ*, we compare and evaluate the costs for $O(n)$ times. Thus, the time complexity of the algorithm is $O(n^2)$ and the algorithm takes $O(1)$ space.

Example 4.3.10 Consider the vq-grams in Figure 4.1(b) and the estimated intersection result sizes of possible vq-gram sets shown in Figure 4.5. Suppose that the query is 'substri' and D_σ^R is {(sub,1), (ubst,2), (bstr,3), (tri,5)}. Among the vq-gram sets with only a vq-gram, {(ubst,2)} has the minimum cost of 29. Thus, we set $OptQ.qset$ and $OptQ.cost$ to {(ubst,2)} and 29 respectively. Then, we enumerate the vq-gram subsets of size 2 including (ubst,2), which are {(sub,1), (ubst,2)}, {(ubst,2), (bstr,3)} and {(ubst,2), (tri,5)}, and compute the cost of each subset. The subset {(sub,1), (ubst,2)} has the minimum cost 36(=21+9+6) among them. However, the minimum cost is larger than the cost of {(ubst,2)} and thus we stop and return {(ubst,2)}. ∎

Up to this point, we assumed that we use MinHash to estimate the intersection result size of intersecting posting lists. In order to find a good vq-gram set even more quickly, we next present an approximate algorithm that uses a crude and simple model using the statistics of posting list sizes only instead of using MinHash technique.

APR-QUICK: Let L_i be the posting list of a vq-gram g_i. Since a positional vq-gram can appear only once in any fixed position of string value in the indexed attribute of every record, all entries in L_is are unique in each posting list. The probability $Pr(g_i, u)$ that a vq-gram g_i appears at the u-th position of string value in a certain record can be formulated as $Pr(g_i, u) = f(g_i)/L$, where $f(g_i)$ is the number of elements in the posting list of the vq-gram g_i and L is the number possible positional vq-grams of all records in the indexed column. The value of $f(g_i)$ is the same as the number of entries in the posting list of g_i, so we can say that $f(g_i) = |L_i|$. Moreover, we assume that $|L_i|$ is proportional to ℓ_i, the number of pages holding L_i in disk. Thus, the probability that a vq-gram g_i appears in the u-th position of a record is $Pr(g_i, u) = |L_i|/L$.

Assuming that every vq-gram occurs at a position in a record independently, we can estimate the probability $Pr(g_i, g_j, u, v)$ that two vq-grams, g_i and g_j, appear together in the u-th and v-th positions, respectively, as the product of $Pr(g_i, u)$ and $Pr(g_j, v)$. Accordingly, the average number of occurrences that two vq-grams, g_i and g_j, appear together with the position difference of k is $L \cdot (f(g_i) \cdot f(g_j))/L^2$ and the average number of records with at least such an occurrence is $(L/\mu) \cdot (f(g_i) \cdot f(g_j))/L^2$, where μ is the average size of a record.

For a given vq-gram set Q, the I/O cost of the processing query with Q will be

$$Cost_s(Q) = \sum_{(g_i,p_i)\in Q} (h + \ell_i - 1) + f_B \left(\frac{L}{\mu} \prod_{(g_i,p_i)\in Q} \frac{|L_i|}{L} \right). \tag{4.8}$$

Let us now discuss how to find a vq-gram set that has the minimal cost based on Equation (4.8). First, we will show below that, among the vq-gram sets with the same size k, the one with minimal cost based on Equation (4.8) can be computed greedily.

Lemma 4.3.11 *Among the vq-gram sets with the same size k, the cost based on Equation (4.8) is minimal when the subset consists of top-k vq-grams whose posting list sizes are the smallest.*

Proof: Let us assume that there is a vq-gram g in a vq-gram set Q with a size of k and let us denote L_g as the posting list of g. If there is a vq-gram $g' \notin Q$ with its posting list $L_{g'}$ and $|L_g| \geq |L_{g'}|$, we can obtain a vq-gram set Q' of size k whose cost

is smaller than Q by substituting g with g' leading to the following inequality

$$\begin{aligned} Cost_s(Q) &= \sum_{g_i \in Q} (h + \ell_i - 1) + f_B\left(\frac{L}{\mu} \prod_{g_i \in Q} \frac{|L_i|}{L}\right) \\ &= \sum_{g_i \in Q/\{g\}} (h + \ell_i - 1) + (h + \ell_g - 1) + f_B\left(\left(\frac{L}{\mu} \prod_{(g_i,p_i) \in Q/\{g\}} \frac{|L_i|}{L}\right) \cdot \frac{|L_g|}{L}\right) \\ &\geq \sum_{(g_i,p_i) \in Q/\{g\}} (h + \ell_i - 1) + (h + \ell_{g'} - 1) \\ &\quad + f_B\left(\left(\frac{L}{\mu} \prod_{(g_i,p_i) \in Q/\{g\}} \frac{|L_i|}{L}\right) \cdot \frac{|L_{g'}|}{L}\right) \qquad (4.9) \\ &= Cost_s(Q'), \end{aligned}$$

where ℓ_g and $\ell_{g'}$ are the number of pages storing posting lists of g and g' respectively. As we mentioned above, ℓ and ℓ' are proportional to $|L_g|$ and $|L_{g'}|$ respectively. Since $f_B(n)$ is a monotonically increasing function such that the inequality $f_B(n) \geq f_B(n')$ holds for $n \geq n'$, the inequality (4.9) can be concluded for $\ell_g \geq \ell_{g'}$ and $L_g \geq L_{g'}$.

Therefore, we can obtain a vq-gram set with a size k, which has a smaller or equal cost, by substituting a vq-gram $g \in Q$ with other vq-gram $g' \notin Q$, when the posting list size of g' is smaller or equal to the posting list size of g. If we repeatedly substitute in this way, we can arrive at a vq-gram set that has the smallest cost, and there remain vq-grams whose posting list sizes are among the k most smallest lengths.∎

Due to Lemma 4.3.11, the cost of the vq-gram set of size k with the minimal I/O cost can be obtained in $O(1)$ time. For a query string of length n and a vq-gram dictionary D with q_{min} and q_{max}, since the size of vq-gram set in D^R_σ is at most $n - q_{min} + 1$, the time complexity of evaluating the cost for all possible sizes is $O(n)$

Function APR-QUICK(D, σ, q_{min}, q_{max})
begin
1. T = MAX-VQGRAM(D, σ, q_{min}, q_{max}, S, E);
2. S = PriorityQueue(T);
3. $Q = \{\}$, $OptQ.cost = \infty$;
4. **while** $|S| > 0$ **do** {
5. $\quad Q = Q \cup \{dequeue(S)\}$;
6. $\quad$ **if** $OptQ.cost > Cost_s(Q)$ **then** {
7. $\quad\quad OptQ.qset = Q$;
8. $\quad\quad OptQ.cost = Cost_s(Q)$;
9. $\quad$ }
10. }
11. **return** $OptQ$;
end

Figure 4.7: APR-QUICK

and the time complexity of sorting the vq-grams in D_σ^R is $O(n \log n)$. Thus, we can conclude the following:

Lemma 4.3.12 *The vq-gram set with the smallest cost can be found in $O(n \log n)$ time where n is the length of the query string.*

We call the algorithm with $O(n \log n)$ time, based on Lemma 4.3.12, *APR-QUICK* which is presented in Figure 4.7. It first lists all vq-grams in D_σ^R for a given query and inserts them into a priority queue S which is a min-heap with the key values of ℓ_i (i.e. the posting list size of i-th vq-gram).

Before the start of iteration, we initialize Q as an empty set and $OptQ.cost$ as ∞. At each iteration, it selects a vq-gram greedily by dequeuing a vq-gram from S which has the minimum size of posting list in S and adds the vq-gram to Q. Since

the vq-gram set that has vq-grams with the top-k smallest posting list sizes has the minimal cost among the possible vq-gram sets of size k, Q in each iteration is the vq-gram set of the minimum cost among the sets of size $|Q|$. Thus, with at most ($|\sigma|$-q_{min}+1) iterations, we can obtain the minimum cost for every size of vq-gram set and can finally obtain the vq-gram set of the minimum cost among all possible subsets of the vq-grams.

Example 4.3.13 Consider the vq-grams in Figure 4.1(b). We assume that L = 1000, μ = 10 and the number of entries in a posting list is the same as the number of pages in the posting list. Suppose that our query is 'substri' and D_σ^R is {(sub,1), (ubst,2), (bstr,3), (tri,5)}. Since the vq-gram whose posting list size is the smallest is 'ubst', the vq-gram subset with the minimum cost among the subsets of size 1 is {(ubst,2)} whose cost is obtained by (9) + (1000/10 * 9/1000) = 9.9. The cost of {(ubst,2), (sub,1)}, which is the vq-gram set of the top-2 vq-grams whose posting list sizes are the smallest, is computed as (9+21) + (1000/10 * 9/1000 * 21/1000) = 30.02. Similarly, we compute the minimum cost of the vq-gram set for every size and select {(ubst,2)} whose cost is the minimum among them. ∎

4.4 Experiments

We now evaluate our substring query optimization algorithms using real life data sets. The objective of this study is to show that the existing algorithms using covering vq-gram sets only are very inefficient compared to our proposed algorithms which consider all subsets of vq-grams, and our proposed optimal and approximate algorithms work well practically.

All experiments were performed on Intel(R) Core(TM)2 Duo CPU 2.66GHz machine with 2GB of main memory running a Linux operating system. All methods were implemented using GCC Compiler of Version 4.1.3.

4.4.1 Implemented Algorithms

We implemented the following algorithms for our performance study.

- **MAX-COVER:** This represents the straightforward query processing algorithm using all vq-grams in D_σ^R, where D_σ^R is defined in Section 4.3.1.
- **MERGE-OPT:** This is the implementation of the extended query processing algorithm of *MergeOpt* in [73] which performs substring matching by searching for the postings from the posting list of every vq-gram in D_σ^R without performing any intersection. It is presented in Section 4.2.
- **OPT-VMINC:** It is an implementation of the algorithm with $O(nq_{max})$ time in Section 4.3.2 which is an extended algorithm of *OPT-MINC* of [65] for the case of using the minimum number of vq-grams covering a query string.
- **OPT-COVER:** This is the implementation of our algorithm with $O(nq_{max})$ time in Section 4.3.2 that finds an optimal covering vq-gram set among covering vq-gram sets.
- **OPT-NAIVE:** It is the implementation of the naive algorithm with enumerating all possible vq-gram sets in Section 4.3.3 that finds an optimal vq-gram set. The time complexity is $O(2^n)$.
- **OPT-QSP:** It represents the implementation of our algorithm with pruning in Section 4.3.3 that finds an optimal vq-gram set among all possible vq-gram sets. The time complexity is $O(2^n)$.
- **APR-DPQ:** This denotes the implementation of our approximate algorithm with $O(n^2)$ time based on dynamic programming in Section 4.3.4.

- **APR-GRQ:** This is our approximate greedy algorithm with $O(n^2)$ time in Section 4.3.4 .

- **APR-QUICK:** This is our approximate greedy algorithm with $O(n \log n)$ time in Section 4.3.4 which uses crude and simple cost estimation without MinHash signatures.

4.4.2 Data Sets

We conducted a series of experiments using the following real-life data sets.

- **Short records:** We used the DBLP titles consisting of 1,000,000 titles collected from DBLP [51] for the short record data set. The maximum and minimum lengths are 516 and 1 bytes respectively, with the average size of titles being 67 bytes. The data size is 65 MB.
- **Long records:** We used the Times articles consisting of 100,000 articles obtained from [79] for the long record data set. The average size of articles is 2578 bytes, with the maximum and minimum lengths of 32407 and 121 bytes respectively. The data size is 224 MB.

4.4.3 Queries Used

We generated one hundred random queries for each query length as follows:

- **Short records:** We collected the terms appearing at DBLP titles for queries and we randomly selected the terms and grouped them by their length. The query length was varied from 4 to 26.

q_{max}	Number of vq-grams
3	82807
4	85095
5	85821
6	86129
7	86255

Figure 4.8: The numbers of vq-grams in dictionaries

- **Long records:** We collected the lines appearing at all Times articles and for each query length, we randomly selected a line and randomly chose the range of query string in the line by truncating its prefix and postfix. The query length was varied from 4 to 32.

As a result of the above generation method, every query in our experiment has at least a matching record in the data sets.

4.4.4 Inverted Indexes Implemented

We built the inverted vq-gram indexes after converting all characters of the strings to be indexed to capital letters. Using the algorithm *Prune* in [53], we built vq-gram dictionaries for each data set and showed the number of vq-grams for the Time articles data set in Figure 4.8, where q_{min} is 3, q_{max} is varied from 3 to 7, and the pruning threshold is 10,000. Since the number of vq-grams in the dictionaries changes slightly when q_{max} is larger than 5, we use $q_{max} = 5$ as the default value for building vq-gram dictionaries. The tries are maintained in main memory and the trie sizes of short and long data sets were 1.3MB and 1.5MB respectively.

Using the vq-gram dictionaries, we built inverted indexes with both disk-based

B+ tree and extendible hashing. Instead of storing vq-grams as key values in B+ tree or extendible hashing, we use the pointers of vq-grams in vq-gram dictionaries as keys to reduce the sizes of indexes. For B+ tree, we want to experiment in the scenario where B+ trees in commercial database systems are used. Thus, since we want to have posting lists in sorted order of (vq-gram-id, record-id, position), we use a multi-column key (vq-gram-id, record-id, position) to store the posting lists in B+ trees. However, for extendible hashing, we use vq-gram-ids only as keys and append the pairs of record-id and position to the posting list for each vq-gram. In each posting list, each element is sorted by record-id first and then by position. However, each record-id is stored only once with the sorted list of positions in its record as illustrated in [13]. Thus, the extendible hashing indexes have much smaller sizes than B+ tree indexes. Furthermore, we can improve the performance of reading the posting lists from inverted indexes by compressing the posting lists with the techniques presented in [75]. If we use the sizes of compressed posting lists for estimating the I/O cost instead of the real sizes, our algorithms can still find the vq-gram set with optimal I/O cost.

Because it takes too long and too much space to build B+ trees for long record data set, we used only samples of the long data set for B+ tree experiments. The sample size was 30% of the original data set, or 70 MB. For extendible hashing, we used the full-sized data sets. The index sizes are generally much larger than the sizes of the original data sets. For example, when we build an inverted index for long record data with $q_{min} = 3$ and $q_{max} = 5$, the size of the B+ tree for the sampled long record data with 70MB is 4GB while that of the extendible hashing index for full-sized long record data with 225MB is 1.6GB only. We also built another small

B+ tree for vq-grams with their MinHash signatures as the associated values in main memory and the size of the B+ tree index was 6MB when the signature length was 50.

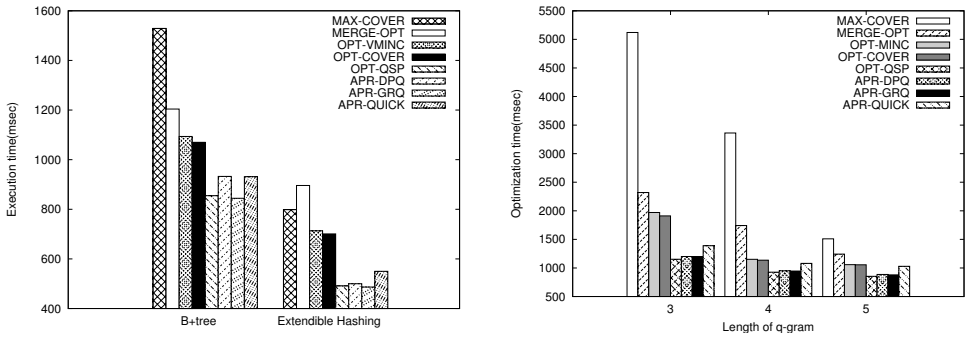
All indexes were built by setting $q_{min} = 3$ and $q_{max} = 3$, 5 and 7, while changing the length of MinHash signatures from 10 to 100. We have chosen the page size of the nodes in all indexes to be 4096 bytes. We implemented our disk-based inverted indexes using our own buffer management without using OS buffers to observe the behaviors of the algorithms more carefully. For each query execution, we cleared buffer space so that buffering effects by previous queries could not help the current query execution. The default buffer size was set to 32MB for our own buffer manager.

4.4.5 Performance Results

We evaluated the proposed algorithms in terms of actual execution times. We conducted experiments to measure the performance of our algorithms with both B+ tree and extendible hashing, while we varied the maximum length of vq-grams, the size of MinHash signatures and the length of query strings. Furthermore, we measured the optimization times for choosing the optimal vq-gram set and the processing times of the chosen plan by the algorithms tested to observe the behaviors of the algorithms. Finally, we conducted experiments for the case where we do not clear the buffer space at the beginning of each query execution to see the buffering effects.

With our performance study, we will establish the following:

- *APR-GRQ*, which is the approximate greedy algorithm, is very fast in terms of



(a) Execution times with B+ Tree and extendible hashing (DBLP titles)

(b) Execution times varying the maximum vq-gram length (Times articles)

Figure 4.9: Execution times of the algorithms

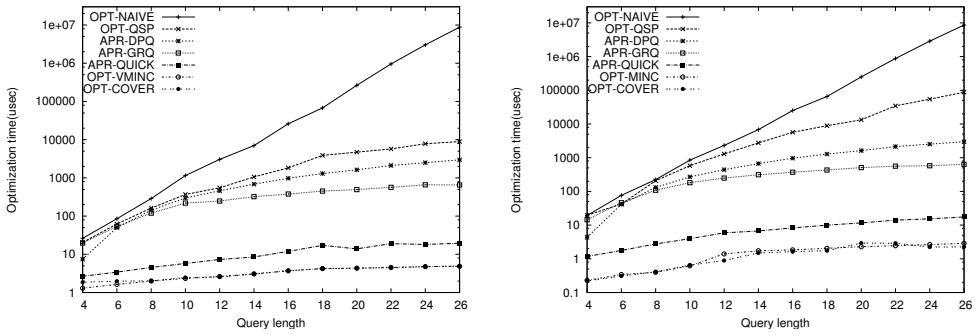
optimization time but the qualities of generated plans are comparable to that of the optimal algorithms.

- *APR-QUICK*, which does not utilize the MinHash signatures, is a good choice if we do not maintain additional statistics for cost estimation.
- *OPT-QSP* is much faster than *OPT-NAIVE* due to efficient pruning of search space. Furthermore, *OPT-QSP* is similar to the approximate algorithms in terms of the optimization time when the sizes of query strings are small. However, *OPT-QSP* becomes slower than the approximate algorithms with long query strings because of the exponential behavior of the search space.
- The length of MinHash signatures does not have that much of an effect on the performance of query processing.

Execution times of B+ tree and Extendible hashing: As we mentioned earlier, the extendible hashing indexes have much smaller sizes than B+ trees and the directories of extendible hashing indexes are maintained in main memory. Thus, for

the same size data, extendible hashing shows faster execution time. The graph of execution times, including both query optimization and processing times using both indexes for DBLP titles data, is shown in Figure 4.9(a). As the figure illustrates, extendible hashing indexes are much faster than B+ trees.

Varying q_{max}: We conducted the experiments by comparing the algorithms while varying the maximum vq-gram length q_{max} from 3 to 7 for both DBLP titles and Times articles using extendible hashing. Since the trends of execution times are similar for both data sets, we show the average execution times of all query lengths from 4 to 32 for Times articles only in Figure 4.9(b). As q_{max} increases, the execution time gradually decreases. The reason is that the size of posting list for a longer vq-gram tends to decrease due to higher selectivity, thus the query processing time gradually decreases. The graph confirms that the query processing times of *MAX-COVER* and *MERGE-OPT* are generally very slow since they read all posting lists of vq-grams in D_{σ}^{R}. Furthermore, the query execution times of the plans generated by *OPT-VMINC* and *OPT-COVER* that consider the covering vq-gram sets only are much slower than the rest of the algorithms exploring all possible vq-gram sets. Thus, our experiment confirms that we should also consider non-covering vq-gram sets to obtain faster execution times. Furthermore, since *APR-QUICK* utilizes a crude and simple method for cost estimation, it is slower than *OPT-QSP*, *APR-DPQ* and *APR-GRQ* which use the MinHash signatures, but is still much faster than the traditional algorithms. The qualities of query execution plans generated by the rest of the algorithms are similar, but *APR-GRQ* is either the fastest or very close to the fastest one.



(a) B+tree (b) Extendable Hashing

Figure 4.10: Optimization time (DBLP titles)

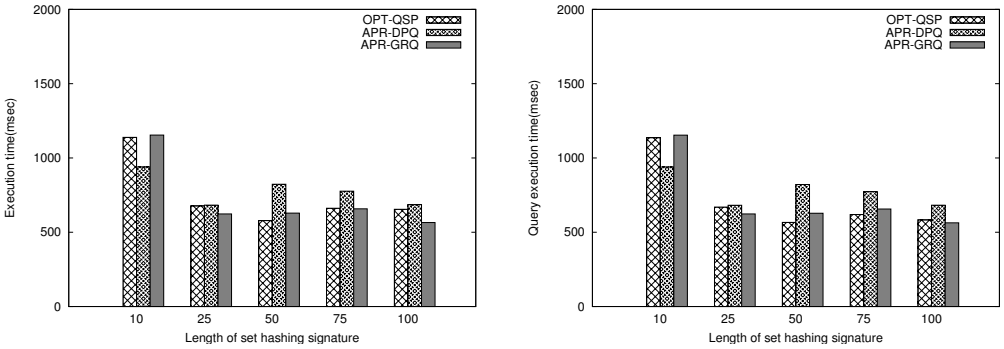
Optimization time: We next show the average optimization times of all algorithms for DBLP titles while varying lengths of queries, using B+ tree and extendible hashing index in Figure 4.10. To show the trends of optimization time, we plotted it on a log scale. As expected, *OPT-NAIVE* shows exponential behavior with increasing query length due to $O(2^n)$ time complexity. We found that the pruning used by *OPT-QSP* is very effective and it performs significantly faster than *OPT-NAIVE* for B+ tree while it is not very effective for extendible hashing. It is because the pruning used in *OPT-QSP* is based on the reading cost of posting lists only. Note that in contrast to B+ tree, there is no I/O cost for traversal of internal nodes in extendible hashing in our implementation and the number of pages for storing the posting lists in extendible hashing is much smaller than that of B+ tree due to compact representation afforded by storing vq-gram IDs in every posting. Thus, the read cost used for pruning in hashing is much smaller than the cost of bringing the result records to check the other selection conditions. (See Equation (4.1) for the components of I/O cost.) For this reason, *OPT-QSP* is not effective when the reading cost of posting lists

is cheap.

As expected, *OPT-QSP* is much slower than the rest of the approximate algorithms. *OPT-VMINC* and *OPT-COVER* are faster than all the other algorithms since they consider the covering vq-gram sets only. Among the rest of the algorithms that consider both covering and non-covering vq-gram sets, *APR-QUICK* is the fastest since it has $O(n \log n)$ time complexity due to its simple cost model while other approximation algorithms have $O(n^2)$ time complexity. *APR-GRQ* is the second fastest and it is also fast for the optimization time. Since *OPT-NAIVE* is too slow for optimization itself, we will discard it for the rest of experiments.

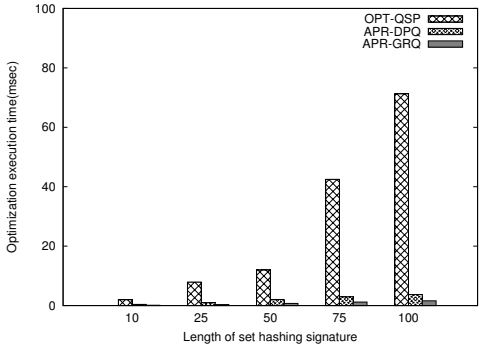
Varying MinHash size: We performed the experiments with varying MinHash signature size from 10 to 100 for Times articles data and DBLP titles with *OPT-QSP*, *APR-DPQ* and *APR-GRQ* which utilize the MinHash signatures for cost estimation. Because the trends of execution times are similar for both data sets, we show the average execution times of all query lengths from 4 to 32 for Times articles using extendible hashing only in Figure 4.11.

As we discussed earlier, the optimization time of *OPT-QSP* is generally a small portion of total execution time due to effective pruning. Thus, we plotted query execution times and query processing times without optimization times separately in Figure 4.11. We found that the size of MinHash signatures does not matter for the quality of optimization and MinHash signatures with small sizes such as 25 are reasonably good for selectivity estimation. However, since optimization requires repetitive invocations of selectivity estimation procedure that uses MinHash signatures, the optimization times of *OPT-QSP* for extendible hashing linearly grow with increasing signature size. Thus, the total execution times of *OPT-QSP* become larger



(a) Total execution time

(b) Query processing time

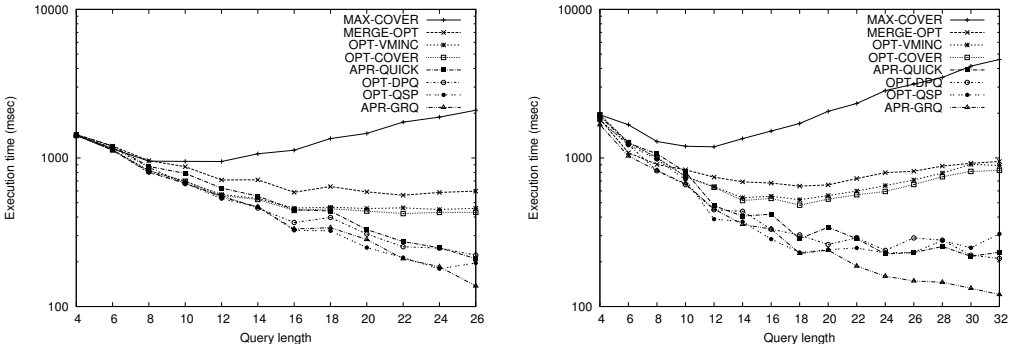


(c) Optimization time

Figure 4.11: Effect of set hashing signature size (Times articles)

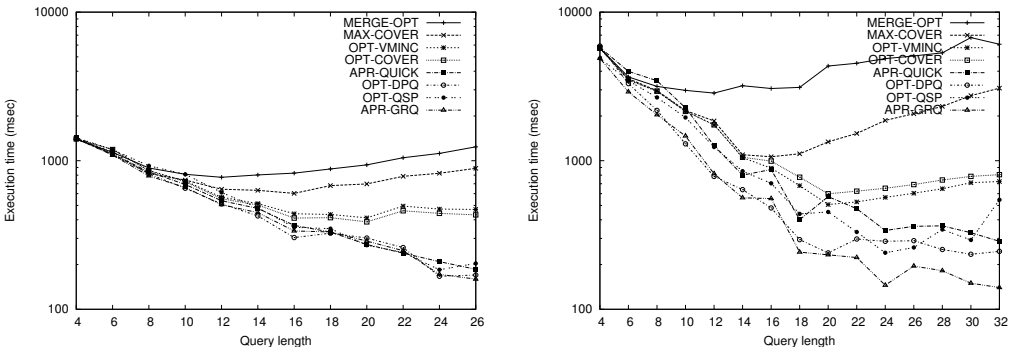
than those of *APR-GRQ* with signature sizes of 75 and 100 even though the query processing times of *OPT-QSP* are the best. For other algorithms, since optimization time is very small compared to execution time, signature size does not matter that much for total execution time.

Varying query length: We plotted the query execution time with varying query length from 4 to 32 for B+ tree and extendible hashing indexes in Figure 4.12 and Figure 4.13 respectively. The log scale was used in the y-axis of graph. The graph for DBLP titles shows that the query execution times by optimization algorithms con-



(a) Execution time (DBLP titles) (b) Execution time (Times articles)

Figure 4.12: Effect of query length (B+tree)



(a) Execution time (DBLP titles) (b) Execution time (Times articles)

Figure 4.13: Effect of query length (extendible hashing)

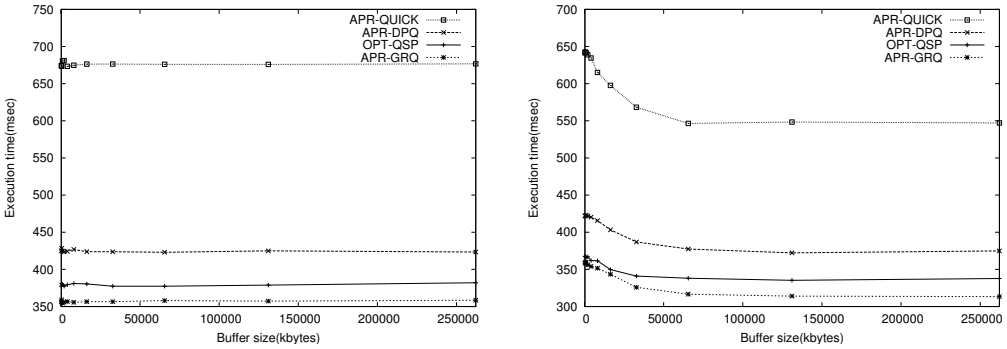
sidering only covering vq-gram sets are much worse than the rest of the algorithms exploring all vq-gram subsets. Notice that as the query length increases, the query execution times of all algorithms except those using covering vq-grams gradually decrease. The reason is that longer query strings have more chance to have vq-grams with higher selectivities. When we use covering vq-gram sets only for query executions, the size of the selected vq-gram set increases as the query lengths become longer and more time is required for query processing. However, since query

processing using all types of vq-gram sets generally utilizes the vq-grams with high selectivities, query execution times decrease as the query length increases.

As shown in Figure 4.9(b), *MAX-COVER* and *MERGE-OPT* are the worst performers with both B+ tree and extendible hash indexes. With B+ tree indexes, since *MERGE-OPT* can search for postings from each posting list by directly accessing the B+ tree with (vq-gram, record ID, position) as keys, the performance of *MERGE-OPT* is shown to be better than *MAX-COVER*. However, with extendible hash indexes, since we do not have a method more efficient than binary search in order to find the postings (record ID, position) from the posting lists in disk, *MERGE-OPT* is much slower than even *MAX-COVER*.

For the algorithms using covering q-gram sets, since *OPT-COVER* considers all covering vq-gram sets, *OPT-COVER* shows better performance than *OPT-VMINC*. The execution times of *APR-GRQ* are the best among the rest of them. Furthermore, *OPT-QSP* is very close to *APR-GRQ*. Thus, the optimization overhead of exploring a larger search space to find an optimal vq-gram set is offset by the good quality of plans generated by *OPT-QSP*. However, *OPT-QSP* becomes slower than the approximate algorithms as the lengths of query strings are larger than 26 because of the exponential behavior of the search space. Since *APR-QUICK* uses crude estimation with its simpler cost model, optimization time is fast but the quality of plans produced is not as good as other algorithms exploring all vq-gram sets. With increasing query length, since there is more chance of having very selective vq-grams, their execution times gradually decrease, except for *MAX-COVER*.

Varying buffer size: All of experiments up to this point flushed the buffer for each query execution. We also experimented without flushing the buffer in each execu-



(a) Buffer flush (extendible hashing) (b) No buffer flush (extendible hashing)

Figure 4.14: Effect of buffering (Times articles)

tion of queries so that the buffering effects by previous queries could be examined. We varied the buffer size from 64KB to 256MB and the experimental results with flushing and without flushing for Times articles data are shown in Figure 4.14. As the buffer size increases, the execution time gracefully decreases as expected. The graphs illustrate that the buffering between queries is very helpful. The graph also confirms that even when we do not clear the buffer at the beginning of query execution, *APR-GRQ* is the best performer.

Chapter 5

Top-k Approximate Substring Matching

The traditional approximate string matching is to find every string from a set of strings whose edit distance to the query string is not larger than a given maximum threshold. The approximate string matching techniques cover diverse applications. For example, it is useful to check whether names or addresses provided by users as input are correct by querying databases to search for the strings within a maximum distance threshold [99]. However, for databases with long strings, the containment search of a query string called *approximate substring matching* is more reasonable.

Two different problem definitions have been introduced for approximate substring matching. In the first definition, the problem is to discover all substrings of each string in a database, whose edit distances to the query string are within the given threshold [54, 85]. Consider a query string 'Jackson' and a database shown in Figure 5.1. If the maximum edit distance threshold is 2, even with s_1='Jackson Pollock' alone, the substrings of 'Jacks', 'Jackso', 'Jackson', 'ackson' and 'ckson' are all

Id	String	Id	String
s_1	Jackson Pollock	s_4	Jacksomville
s_2	Jakob Pollack	s_5	Jakson Pollack
s_3	Jason Polock	s_6	Mackson Polock

Figure 5.1: An example of a string database D

included in the query result.

Another popular definition of the approximate substring matching is to retrieve every string s whose *substring edit distance* to the query string σ is at most the given threshold, where substring edit distance between σ and s is the smallest edit distance among the ones between σ and all substrings in s [48, 83]. For example, let us reconsider the string database in Figure 5.1 and the query string 'Jackson'. If the maximum threshold is 2, the approximate substring matches are {'Jackson Pollock', 'Jason Polock', 'Jacksomville', 'Jakson Pollack', 'Mackson Polock'}.

All the studies of approximate string or substring matching mentioned so far require a user to provide a maximum distance threshold. However, since it is very difficult to know this threshold in advance, it is more practical to compute the most similar k strings or substrings without the need to specify a distance threshold. For example, consider the string data in Figure 5.1. Suppose that the query string σ is 'Jackson' and $k = 3$. Since s_1 has the substring which is exactly σ, its substring edit distance to σ is 0. For s_4 and s_5, the substring edit distances to σ are 1 since s_4 and s_5 have 'Jacksom' and 'Jakson' respectively. For other strings, the substring edit distances to σ are at least 1. Thus, the top-3 approximate substring matches are {'Jackson Pollock', 'Jacksomville', 'Jakson Pollack'}.

In this chapter, we propose efficient algorithms for top-k approximate substring

matching problem which find the top-k strings whose substring edit distances to the query string are the smallest among all strings in a database. Contrast to traditional approximate approximate substring matching, top-k approximate matching does not require a user to provide a distance threshold. To the best of our knowledge, no existing work has addressed the top-k approximate substring matching problem and our algorithms presented here are the first work for the problem. In the following, we first define the problem of top-k approximate substring matching.

Problem definition: For two strings s_i and s_j, we assume that s_i can be transformed to s_j by applying repeatedly the three operations: insertion, deletion and substitution. For two strings s_i and s_j, the *edit distance* [50] between s_i and s_j, which is denoted by $d(s_i, s_j)$, is defined as the minimum number of operations needed to transform s_i to s_j (or s_j to s_i). Furthermore, the *substring edit distance* [76] from a query string σ to another string s, which is represented by $d_{sub}(s, \sigma)$, is the minimum among the edit distances between σ and every substring of s.

We now present the definition of the *top-k approximate substring matching*.

Definition 5.0.1 Given a collection of strings D and a query string σ, the top-k approximate substring matching is to find the k strings $TopS(k, \sigma) = \langle s_1, s_2, ..., s_k \rangle$ in D which are an ordered sequence of k strings satisfying the following conditions:

- $d_{sub}(s_1, \sigma) \leq d_{sub}(s_2, \sigma) \leq \cdots \leq d_{sub}(s_k, \sigma)$ holds.
- For every $s_j \in D$ such that $s_j \notin TopS(k, \sigma)$, we have $d_{sub}(s_k, \sigma) \leq d_{sub}(s_j, \sigma)$.

Example 5.0.2 Consider the strings shown in Figure 5.1. Suppose that the query string σ is 'Jackson' and we are interested in the top-3 approximate substring matches $TOP_3(\sigma)$. Since s_1 includes the substring which is exactly the same as σ, the substring edit distance $d_{sub}(s_1, \sigma)$ is 0. For the string s_4, $d_{sub}(s_4, \sigma)$ is 1 since s_4 has a

substring 'Jacksom'. Similarly, $d_{sub}(s_5, \sigma)$ is 1 since the string s_5 contains 'Jakson'. For other strings, the substring edit distances to σ are at least 1. Thus, we get $TOP_3(\sigma)$=$\{s_1$,s_4,$s_5\}$. ∎

5.1 The Lower Bounds of Substring Edit Distances

We first present how to compute the lower bounds of substring edit distances by utilizing q-gram properties to identify the strings in D which do not need to compute actual substring edit distances. For correctness and efficiency, we develop our techniques to guarantee *no false dismissals* and *few false positives*, respectively. To achieve these goals, we devise several filtering methods.

The lower bound of edit distance: We denote the lower bound of edit distance $d(s, \sigma)$ between a string s and a query string σ by $\ell o(d(s, \sigma))$. The following lemma allows us to compute $\ell o(d(s, \sigma))$.

Lemma 5.1.1*Given a query string σ and a string s, if s has at most c common q-grams with σ, the edit distance between $d(s, \sigma)$ satisfies the following inequality.*

$$d(s, \sigma) \geq \lceil (\max(|\sigma|, |s|) - q + 1 - c)/q \rceil$$

In other words, $\ell o(d(s, \sigma)) = \lceil (\max(|\sigma|, |s|) - q + 1 - c)/q \rceil$.

Proof: We split into two cases: $|s| \leq |\sigma|$ and $|s| > |\sigma|$.

- **when** $|s| \leq |\sigma|$**:** Since we have ($|\sigma|-q+1$) q-grams in σ and we know that the string s shares at most c q-grams with σ, there are at least (($|\sigma|-q+1$)$-c$) mismatching q-grams from σ to s. Consider a string s_1 and another string s_2 which is obtained by applying a single edit operation to s_1. Then, we have at most q mismatching q-grams from the string s_1 to s_2. Since we have (($|\sigma|-q+1$)$-c$)

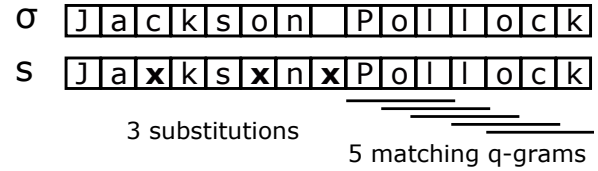


Figure 5.2: The lower bound of $d(s, \sigma)$

mismatching q-grams from σ to s, we need at least $\lceil(|\sigma|-q+1-c)/q\rceil$ edit operations to convert σ to s. Thus, a lower bound of $d(s, \sigma)$ is $\lceil(|\sigma|-q+1-c)/q\rceil$ which can obtained with at least $(|\sigma|-|s|)$ deletion operations.

- **when** $|s| > |\sigma|$**:** For the case of $|s|>|\sigma|$, if we simply switch s and σ in the above proof, a lower bound of $d(s, \sigma)$ becomes $\lceil(|s|-q+1-c)/q\rceil$.

Considering both cases, we can conclude that $d(s, \sigma) \geq \lceil(\max(|\sigma|, |s|)-q+1-c)/q\rceil$.

∎

Example 5.1.2 Consider a query string σ='Jackson Pollock'. Since the length of σ is 15, the number of 3-grams in σ is 13. For a string s='JaxksxnxPollock' which has 5 matching 3-grams (i.e., 8 mismatching 3-grams) with σ, a lower bound of $d(s, \sigma)$ is $\lceil(13-5)/3\rceil$=3 due to Lemma 5.1.1. Note that the actual edit distance between s and σ is 3 as shown in Figure 5.2(a) since we need three substitutions at the positions of 3, 6 and 8 in σ. ∎

The lower bounds of substring edit distance: We now present the lemma below which allows us to compute a lower bound of the substring edit distance $d_{sub}(s, \sigma)$ between a string s and the query string σ. Remember that $s[i, j]$ denote the substring of s which starts from position i and ends at position j.

Lemma 5.1.3*Consider a query string σ and a string s. Let c be the number of common q-grams between σ and s. Furthermore, let c_i be the number of common q-grams*

between σ and $s[i, i + |\sigma| - 1]$. Then, the lower bound of $d_{sub}(s, \sigma)$, represented by $\ell o(d_{sub}(s, \sigma))$, is

$$\ell o(d_{sub}(s,\sigma)) = \begin{cases} \lceil (|\sigma| - q + 1 - c)/q \rceil, & \text{if } |s| < |\sigma|, \\ \min_{1 \le i \le |s|-|\sigma|+1} \lceil (|\sigma| - q + 1 - c_i)/q \rceil, & \text{otherwise.} \end{cases} \quad (5.1)$$

Proof: If $|s| < |\sigma|$, σ cannot be a substring of s. Thus, the lower bound of $d_{sub}(s, \sigma)$ is the same as the lower bound of $d(s, \sigma)$ which is actually $\lceil (|\sigma|-q+1-c)/q \rceil$ by Lemma 5.1.1.

When $|s| \geq |\sigma|$, in order to compute the lower bound of $d_{sub}(s, \sigma)$, we can enumerate every substring s' of s and compute the lower bound of edit distance between σ and every substring s'.

Consider a substring $s[i, i+\ell-1]$ of length ℓ. If ℓ is larger than $|\sigma|$, the lower bound of $d(s[i,i+\ell-1], \sigma)$, which we denote by $\ell o(d(s[i,i+\ell-1), \sigma))$, is at least $\ell o($ $d(s[i,i+|\sigma|-1], \sigma))$. It is because the lower bound derived by Lemma 5.1.1 monotonically grows with increasing the length of string s. Since computing substring edit distance requires to find the substring with the minimum edit distance to σ, we need not to consider the substrings of s whose lengths are larger than $|\sigma|$. Thus, $\ell o(d_{sub}(s, \sigma))$ is the minimum one among $\ell o(d(s[i,i+|\sigma|-1], \sigma))$s with $1 \le i \le |s|-|\sigma|+1$.

∎

Corollary 5.1.4 *For a string s and the query string σ with $|s| \geq |\sigma|$, $\ell o(d_{sub}(s, \sigma))$ is the lower bound of the edit distance between σ and the substring in s with the length $|\sigma|$ which has the maximum number of common q-grams with σ among all substrings $s[i, i + |\sigma| - 1]$s for $1 \le i \le |s|-|\sigma|+1$.*

Proof: Due to Lemma 5.1.3, $\ell o(d_{sub}(s, \sigma))$ is the minimum among $\lceil (|\sigma| - q + 1 - c_i)/q \rceil$s computed with $1 \le i \le |s|-|\sigma|+1$ where c_i is the number of common q-grams

between σ and the substring $s[i, i+|\sigma|-1]$. Since $\lceil(|\sigma| - q + 1 - c_i)/q\rceil$ becomes the smallest when c_i is the largest, $\ell o(d_{sub}(s, \sigma))$ is $\ell o(d(s[i, i + |\sigma| - 1], \sigma))$ where $s[i, i + |\sigma| - 1]$ has the largest number of common q-grams with σ. ∎

By Corollary 5.1.4, to calculate $\ell o(d_{sub}(s, \sigma))$, we can consider the substrings of s with the size $|\sigma|$ only and find the maximum number of common q-grams between σ and every substring $s[i, i+|\sigma|-1]$. If we simply count the common q-grams between σ and each substring $s[i, i+|\sigma|-1]$ with $1 \leq i \leq |s|-|\sigma|+1$, it takes $O(|\sigma| \cdot |s|)$ time.

To compute $\ell o(d_{sub}(s, \sigma))$ more quickly, we can compute non-zero c_is only by maintaining a sliding window in the common positional q-gram list of σ and s. We refer to this algorithm as *CALC-LB* and it takes only $O(|s|)$ time.

The CALC-LB algorithm: To compute $\ell o(d_{sub}(s, \sigma))$, instead of computing c_i by simply counting the number of common q-grams between σ and every substring $s[i, i+|\sigma|-1]$ with $1 \leq i \leq |s|-|\sigma|+1$, we computes non-zero c_is only by maintaining a sliding window in the common positional q-gram list of σ and s.

Let us consider the list $R = \langle(g_1, p_1), \ldots (g_n, p_n)\rangle$ of common positional q-grams between σ and s in the increasing order of positions. The position p_i of each q-gram g_i in the list is assigned to the position in s of the q-gram g_i. To count the common q-grams between σ and each substring $s[i, i+|\sigma|-1]$, we scan each positional q-gram in R one by one with maintaining a sliding window $[b, e]$ of which b and e represent the indexes of beginning and ending positional q-grams in R respectively. When examining the j-th positional q-gram (g_j, p_j), we want to move forward the sliding window so that it contain the matching q-grams between σ and $s[p_j + q - |\sigma|, p_j + q - 1]$. Note that the right most q-gram in the sliding window becomes g_j since the last character of g_j is located at the position $p_j + q - 1$. Thus, we first set e to j and

increase b until the the b-th q-gram g_b is positioned within the sliding window. Now we can simply use $(e-b+1)$, which represents $c_{p_j+q-|\sigma|}$, as the number of common q-grams between $|\sigma|$ and $s[p_j+q-|\sigma|, p_j+q-1]$.

We refer to the algorithm to compute the lower bound of $d_{sub}(s,\sigma)$ as *CALC-LB* and present the pseudocode in Figure 5.3. When we start *CALC-LB*, we let $b=e=1$. Thus, we consider the substring $s[p_1+q-|\sigma|, p_1+q-1]$, which includes only a common q-gram g_1 between σ and $s[p_1+q-|\sigma|, p_1+q-1]$, and the number of common q-grams $c_{p_1+q-|\sigma|}$ becomes 1.

At the beginning of examining the j-th q-gram in the list R, we set e to j and also increase b until the b-th q-gram's position p_b becomes at least $p_j+q-|\sigma|$. Then, with the j-th q-gram, $c_{p_j+q-|\sigma|}$, which is the number of common q-grams between σ and $s[p_j+q-|\sigma|, p_j+q-1]$, is $(e-b+1)$. By repeating this step until we reach the last q-gram in R, we can compute the maximum value among non-zero c_is with $O(|s|)$ time.

Previously, we could compute the lower bound of $d_{sub}(s,\sigma)$ using the number of common q-grams between the query string σ and the string s. However, we next show how to compute the lower bound of $d_{sub}(s,\sigma)$ using the number of mismatching q-grams from σ to s (i.e., the q-grams occurring in σ but not in s) if we do not have positional information.

Lemma 5.1.5 *Consider a query string σ and a string s. If the number of mismatching q-grams from σ to s is at least m, we have $d_{sub}(s,\sigma) \geq \lceil m/q \rceil$.*

Proof: When a string s_2 is obtained by applying a single edit operation on a string s_1, we have at most q mismatching q-grams from s_1 to s_2. Since at least m q-grams

Function CALC-LB (R, ℓ)
R : a list of positional q-grams $\langle (g_1, p_1), \ldots (g_n, p_n) \rangle$,
ℓ : the length of query string.
begin
1. $b = 1$, $max = 0$;
2. **for** $j = 1$ to $|R|$ **do** {
3. $\quad e = j$;
4. $\quad$ **while** ($p_b < p_j + q - \ell$) **do**
5. $\quad\quad b = b + 1$;
6. $\quad$ **if** $max < e - b + 1$ **then**
7. $\quad\quad max = e - b + 1$;
8. }
9. **return** $\lceil (\ell - q + 1 - max)/q \rceil$;
end

Figure 5.3: The CALC-LB algorithm

in σ do not appear in s, we need at least $\lceil m/q \rceil$ edit operations to convert σ to s (i.e., $\ell o(d(s, \sigma)) = \lceil m/q \rceil$). Furthermore, for every substring s' of the string s, since the number of mismatching q-grams from σ to s' is at least that of mismatching q-grams from σ to s, we can claim that the number of mismatching q-grams from σ to s' is at least m. Thus, we have $d(s', \sigma) \geq \lceil m/q \rceil$ for every substring s' of s and we can conclude $d_{sub}(s, \sigma) \geq \lceil m/q \rceil$. ∎

Consider a subset $G' \subseteq G$ where G is the set of all possible q-grams from σ. Assume that we partition D into $D^+_{G'}$ and $D^-_{G'}$ so that $D^+_{G'}$ is the set of strings in D which share at least a q-gram in G' and $D^-_{G'}$ includes the rest of strings in D (i.e., $D^-_{G'} = D - D^+_{G'}$). Let $d_{sub}(D, \sigma)$ represent the smallest substring edit distance between σ and every string s in D. We denote the lower bound of $d_{sub}(D, \sigma)$ by

$\ell o(d_{sub}(D, \sigma))$.

The lower bounds of $d_{sub}(D^-_{G'}, \sigma)$: We can compute a lower bound of the substring edit distances between σ and every string in $D^-_{G'}$ by the following lemma.

Lemma 5.1.6 *Assume that we have a set of strings D and a query string σ. For each subset $G' \subseteq G$ where G is the q-gram set of σ, $d_{sub}(D^-_{G'}, \sigma) \geq \lceil |G'|/q \rceil$.*

Proof: Due to Lemma 5.1.5, for every string s in $D^-_{G'}$ which does not include any q-gram in G', $d_{sub}(s, \sigma) \geq \lceil |G'|/q \rceil$. Thus, $d_{sub}(D^-_{G'}, \sigma) \geq \lceil |G'|/q \rceil$ ∎

While the lower bound of obtained by Lemma 5.1.6 is effective, it does not take advantage of q-gram positions in G'. In other words, we assumed that the mismatching q-grams between σ and each string in $D^-_{G'}$ are always overlapped. However, if we choose the q-grams for G' such that every q-gram in G' does not overlap to any other q-gram in G', we can obtain a tighter value of $\ell o(d_{sub}(D^-_{G'}, \sigma))$.

Lemma 5.1.7 *Consider a query string σ. Let G be the set of all q-grams from σ. If we select a subset $G' \subseteq G$ which does not have any q-gram with overlapped position in σ to another q-gram in G', we have $d_{sub}(D^-_{G'}, \sigma) \geq |G'|$.*

Proof: Intuitively, if G' has the q-grams which do not overlap to other q-grams in G' at any positions in σ, we need at least an edit operation to make each mismatching q-gram in G'. Thus, for every string s in $D^-_{G'}$, we need at least $|G'|$ edit operations to transform σ to any substring of s.∎

Example 5.1.8 Suppose we have a query string σ='Jackson' and two 3-grams 'Jac' and 'kso', which are the 3-grams without any overlapped positions in σ, are chosen for G'. The strings in $D^-_{G'}$ are the strings which do not share even one of those two 3-grams with σ. Since the mismatching 3-grams 'Jac' and 'kso' do not overlap to each

other, to transform σ to every substring of each string in $D^-_{G'}$, we need at least 2 edit operations (an edit operation on each of 'Jac' and 'kso') in σ. Note that we have $d_{sub}(D^-_{G'}, \sigma) \geq 2$ by Lemma 5.1.7.

Let us consider G'' consisting of 'Jac' and 'cks' which overlap at the third character in σ. For a string s which is the same with σ except the position of overlapped character (e.g., 'Jakkson'), the substring edit distance $d_{sub}(s, \sigma)$ is 1. Since there may exist such a string s in $D^-_{G''}$, the lower bound of $d_{sub}(D^-_{G''}, \sigma)$ should be at most 1. Note that we have $d_{sub}(D^-_{G''}, \sigma) \geq 1$ due to Lemma 5.1.6. ∎

If we fix the size of $G' \subseteq G$, we obtain the tightest lower bound $\ell o(d_{sub}(D^-_{G'}, \sigma))$ when G' is selected without overlapping q-grams.

Corollary 5.1.9 *Consider a query string σ and let G be the set of q-grams in σ. We obtain the tightest lower bound $\ell o(d_{sub}(D^-_{G'}, \sigma))$ among every possible $G' \subseteq G$ when the size of G' is $\lfloor |\sigma|/q \rfloor$.*

Proof: This is because we cannot select a set of q-grams with a larger size than $\lfloor |\sigma|/q \rfloor$ each of which does not overlapped to other q-grams in the set. ∎

We now introduce the following lemma to use Lemma 5.1.7 effectively for top-k approximate substring matching.

Lemma 5.1.10*Consider a set of strings D, a query string σ and the q-gram set G of σ. Assume that we partitioned D into $D^+_{G'}$ and $D^-_{G'}$ for a subset $G' \subseteq G$. While examining every string in $D^+_{G'}$, if there are at least k strings whose substring edit distances to σ are at most $\ell o(d_{sub}(D^-_{G'}, \sigma))$, we can find the top-$k$ approximate substring matches in D by examining the remaining strings in $D^+_{G'}$ only with ignoring the strings in $D^-_{G'}$.*

Proof: Let s_k denote the string with the k-th smallest substring edit distance $d_{sub}(s_k, \sigma)$ while examining every string in $D^+_{G'}$ one by one. Since $d_{sub}(s_k, \sigma) \leq \ell o(d_{sub}(D^-_{G'}, \sigma))$,

for every string $s' \in D^{-}_{G'}$, $d_{sub}(s', \sigma)$ cannot get smaller than $d_{sub}(s_k, \sigma)$ and we can not have any string of the top-k approximate substring matches in $D^{-}_{G'}$. ∎

5.2 Top-k Approximate Substring Matching Algorithms

We first introduce the brute-force algorithm *TopK-NAIVE* which blindly examines every string s in D and computes substring edit distances $d_{sub}(s, \sigma)$ one by one. To reduce the number of substring edit distances to compute, we propose the algorithm *TopK-LB* which checks the possibility of top-k approximate substring matches first by utilizing the quickly computable lower bound $\ell o(d_{sub}(s, \sigma))$ and then computes $d_{sub}(s, \sigma)$ only when necessary. We next present the algorithm *TopK-SPLIT* which allows us even not to calculate the lower bounds $\ell o(d_{sub}(s, \sigma))$ for some strings s in D by partitioning data D into groups and taking advantages of the lower bound of substring edit distances between σ and all strings appearing in a group.

5.2.1 TopK-NAIVE

The naive algorithm examines every string s in a set D of strings and computes the substring edit distance $d_{sub}(s, \sigma)$. To find the top-k approximate substring matches in D, we maintain a max-heap H_{TopK} storing the k strings s' with the smallest $d_{sub}(s', \sigma)$s which are used as the keys in the max-heap H_{TopK}. If the size of H_{TopK} is less than k, we just insert the string s to H_{TopK}. Otherwise, we check whether $d_{sub}(s, \sigma)$ is smaller than $d_{sub}(s_R, \sigma)$ of the string s_R at the root of H_{TopK} (i.e. whether $d_{sub}(s, \sigma)$ is not smaller than the k-th smallest substring edit distance so far). If it is satisfied, we delete the string at the root s_R of H_{TopK} and insert the string s to H_{TopK}. If not, we move to the next string s and repeat the above step un-

Function TopK-LB (D, k, σ)
begin
1. H_{TopK} = an empty max-heap storing ⟨dist, str⟩;
2. $G = \{\sigma[i,i+q-1] | 1 \leq i \leq |\sigma|-q+1\}$;
3. **for each** string s in D **do** {
4. $R = \{(s[i,i+q-1], i) \mid \forall i$ s.t.$1 \leq i \leq |s|-q+1$, $s[i,i+q-1] \in G\}$;
5. LB = CALC-LB(R, $|\sigma|$);
6. **if** $|H_{TopK}| < k$ or $LB < H_{TopK}$.getMax().dist **then** {
7. $d = d_{sub}(s, \sigma)$;
8. **if** $|H_{TopK}| < k$ **then** H_{TopK}.insert(⟨d,s⟩);
9. **else if** H_{TopK}.getMax().dist $> d$ **then** {
10. H_{TopK}.insert(⟨d,s⟩);
11. H_{TopK}.deleteMax();
12. }
13. }
14. }
15. **return** H_{TopK};
end

Figure 5.4: The TopK-LB algorithm

til we encounter the last string in D. We refer to the naive algorithm as *TopK-NAIVE*.

5.2.2 TopK-LB

Similar to *TOPK-NAIVE*, we scan all strings in D. However, for each string s in D, we generate the list of common positional q-grams $R = \langle (g_1, p_1), \ldots (g_n, p_n) \rangle$ between σ and s in increasing order of positions where the position p_i of each (g_i, p_i) in R is the position in s at which the q-gram g_i appears. Then, we can compute the lower bound $\ell o(d_{sub}(s, \sigma))$ by invoking *CALC-LB* which utilizes Lemma 5.1.3. If $\ell o(d_{sub}(s, \sigma))$ is not smaller than the k-th smallest substring edit distance so far, we do not calculate

the actual value of $d_{sub}(s, \sigma)$ since s cannot be a string in the top-k approximate substring matches. We call the algorithm *TopK-LB* and present the pseudocode of *TopK-LB* in Figure 5.4.

Example 5.2.1 Consider the set of strings D in Figure 5.1. Suppose the query string σ='Jacksen' of length 7 and we are interested in the top-2 approximate matches. The first two strings s_1 and s_2, whose substring edit distances to σ are 1 and 4 respectively, are inserted into the heap H_{TopK}. The next string s_3='Jason Polock' does not have any common 3-gram with σ. Thus, the lower bound $\ell o(d_{sub}(s_3, \sigma))$ is 2 $(=\lceil((7-3+1)-0)/3\rceil)$ according to Lemma 5.1.3. Since the second smallest substring edit distance in H_{TopK} is 4 and $\ell o(d_{sub}(s_3, \sigma))$ is smaller than 4, we have to compute $d_{sub}(s_3, \sigma)$, which turns out 3, and insert $(s_3, 3)$ into H_{TopK}. Now we have $\{(s_1, 1), (s_3, 3)\}$ in H_{TopK} and the second smallest substring edit distance so far becomes 3.

With s_4='Jacksomville', the common positional 3-gram list R is {('Jac',1), ('ack', 2), ('cks',3)} and the lower bound $\ell o(d_{sub}(s_4, \sigma))$ becomes 1. Since $\ell o(d_{sub}(s_4, \sigma))$ is smaller than the second smallest substring edit distance in H_{TopK}, we have to compute $d_{sub}(s_4, \sigma)$ which is 2 and $(s_4, 2)$ is added to H_{TopK}. Then, the second smallest substring edit distance so far becomes 2. For s_5='Jakson Pollack', since we have no common 3-grams with σ, $\ell o(d_{sub}(s_5, \sigma))$ is 2 which is at least the second smallest substring edit distance so far. Thus, we can skip computing expensive $d_{sub}(s_5, \sigma)$. Finally, with s_6='Mackson Polock', $\ell o(d_{sub}(s_5, \sigma))$ is 1 and we should compute $d_{sub}(s_5, \sigma)$. In summary, we calculated the substring edit distances with 5 strings out of 6 strings. ∎

5.2.3 TopK-SPLIT

Let G be the q-gram set of σ and let G' be a subset of G (i.e., $G' \subseteq G$). We will discuss how to select G' in the next section and for the time being, we will assume that G' is provided by a function named *BEST-G'*.

Function TopK-SPLIT (D, k, σ)
begin
1. H_{TopK} = an empty max-heap storing ⟨dist, str⟩;
2. $G = \{\sigma[i,i+q-1] | 1 \leq i \leq |\sigma|-q+1\}$;
3. G' = BEST-G'(σ, k, $|D|$);
4. $\tau = |G'|$;
5. $prune = false$;
6. **for each** string s in D **do** {
7. $R = \{(s[i,i+q-1], i) \mid \forall i$ s.t.$1 \leq i \leq |s|-q+1$, $s[i,i+q-1] \in G\}$;
8. **if** there exists a $(g_i, p_i) \in R$ with $g_i \in G'$ **then** $part$ = '+';
9. **else** $part$ = '−';
10. **if** $prune = false$ or $part$ = '+' **then** {
11. LB = CALC-LB(R, $|\sigma|$);
12. **if** $|H_{TopK}| < k$ or $LB < H_{TopK}$.getMax().dist **then** {
13. $d = d_{sub}(s, \sigma)$;
14. **if** $|H_{TopK}| < k$ **then** H_{TopK}.insert(⟨d,s⟩);
15. **else if** H_{TopK}.getMax().dist $> d$ **then** {
16. H_{TopK}.insert(⟨d,s⟩);
17. H_{TopK}.deleteMax();
18. }
19. **if** H_{TopK}.getMax().dist $\leq \tau$ **then** $prune = true$;
20. }
21. }
22. }
23. **return** H_{TopK};
end

Figure 5.5: The TopK-SPLIT algorithm

We conceptually divide the strings in D into $D^+_{G'}$ and $D^-_{G'}$ where $D^+_{G'}$ is the set of strings in D each of which has at least a q-gram in G' and $D^-_{G'}$ is ($D - D^+_{G'}$). Then,

we can calculate the lower bound of $d_{sub}(D^-_{G'}, \sigma)$ which is $|G'|$ due to Lemma 5.1.7. While scanning each string s, we can check easily whether the string s belongs to $D^+_{G'}$ or $D^-_{G'}$ by checking the q-grams in s.

Similar to *TopK-LB*, when we examine every string s in D one by one, we skip computing $d_{sub}(s, \sigma)$ if $\ell o(d_{sub}(s, \sigma))$ is at least the k-th smallest substring edit distance so far. However, unlike *TopK-LB*, we perform the following additional step for each string.

As soon as the k-th smallest substring edit distance so far becomes at most the lower bound of $d_{sub}(D^-_{G'}, \sigma)$ (i.e., $d_{sub}(s_R, \sigma) \geq \ell o(d_{sub}(D^-_{G'}, \sigma))$ where the string s_R at the root of H_{TopK}), we can skip even computing the lower bounds $\ell o(d_{sub}(s, \sigma))$ for unseen strings, which belonging to $D^-_{G'}$, due to Lemma 5.1.10. However, for the strings belonging to $D^+_{G'}$, we still perform the same step for substring edit distance computations. We call this algorithm *TopK-SPLIT*.

We show the pseudocode of *TopK-SPLIT* algorithm in Figure 5.5. We first choose a q-gram set G' by invoking *BEST-G'*. Due to Lemma 5.1.7, we obtain the lower bound of $d_{sub}(D^-_{G'}, \sigma)$ as $|G'|$ which is represented by τ. We also set $prune$ to $false$ initially representing that the k-th smallest substring edit distance so far exceeds τ. Similar to *TopK-LB*, while we scan each string s in D, if $LB = \ell o(d_{sub}(s, \sigma))$ computed by *CALC-LB* is smaller than the k-th smallest substring edit distance so far, we compute the actual value of $d_{sub}(s, \sigma)$. Unlike *TopK-LB*, if the k-th smallest substring edit distance becomes at most τ, we set $prune$ to $true$ so that we skip the computation of $d_{sub}(s, \sigma)$ for the unseen strings $s \in D^-_{G'}$ by Lemma 5.1.10.

Example 5.2.2 Consider the strings D in Figure 5.1. Consider the query string $\sigma =$ 'Jacksen' for top-2 approximate substring matching using 3-grams. Suppose that the

3-gram set G' is {'Jac', 'kse'}. Due to Lemma 5.1.7, the lower bound of $d_{sub}(D^-_{G'}, \sigma)$ is 2.

First, the strings s_1 and s_2 are inserted into H_{TopK} whose substring edit distances to σ are 1 and 4 respectively. With the next string s_3='Jason Polock', the lower bound of its substring edit distance to σ is smaller than the second smallest substring edit distance so far and we compute $d_{sub}(s_3, \sigma)$. For s_4='Jacksomville', we also compute $d_{sub}(s_3, \sigma)$ and the second smallest substring edit distance in H_{TopK} becomes 2.

Since we now have 2 strings whose substring edit distances are at most the lower bound of $d_{sub}(D^-_{G'}, \sigma)$ which is 2, we can find the top-2 approximate substring matches by considering only the strings sharing at least a 3-gram in G' due to Lemma 5.1.10. Thus, for the strings s_5 and s_6, we can ignore them because they do not have any common 3-gram with G'. In summary, we could find the top-2 approximate substring matches by calculating the substring edit distances with 4 strings only. ∎

5.3 Selecting a Proper G' among all $G' \subseteq G$

In previous section, we assumed that $G' \subseteq G$ is provided by the function *BEST-G'* for *TopK-SPLIT*. Now, we present the algorithm *BEST-G'* which selects a proper G' among all subsets of G. We first introduce the method to estimate the computational costs for finding top-k approximate substring matching with *TopK-SPLIT*. Then, we provide an algorithm *SELECT-G'(ρ)* which finds G' of size ρ such that the number of strings in D including a q-gram in G' is the smallest. Finally, we present the algorithm *BEST-G'* to select G' which minimizes our proposed computational cost formula among G's of all possible sizes using *SELECT-G'(ρ)*.

Average number of substring edit distance computations: In *TopK-SPLIT*, we scan every string s in D one by one and compute the substring edit distance $d_{sub}(s, \sigma)$ un-

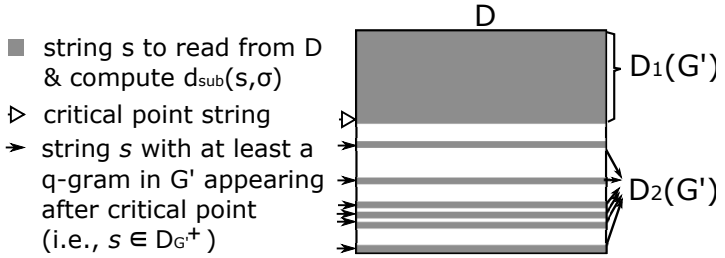


Figure 5.6: The computational cost

til we find k strings with the substring edit distances of at most the lower bound of $d_{sub}(D^-_{G'}, \sigma)$ which is $\ell o(d_{sub}(D^-_{G'}, \sigma))$ (i.e., the condition in Lemma 5.1.10 is satisfied). We refer to the k-th string with substring edit distance at most $\ell o(d_{sub}(D^-_{G'}, \sigma))$ as *critical point string*. Then, we compute $d_{sub}(s, \sigma)$ for the rest of strings s in D if the string s contains a q-gram in G'.

As we illustrate in Figure 5.6, let $D_1(G')$ be the number of strings to examine in D until we reach the critical point string (i.e., until there are k strings whose substring edit distances are at most $\ell o(d_{sub}(D^-_{G'}, \sigma))$). Let $D_2(G')$ be the number of strings in $D^+_{G'}$ that appear after the critical point string in D. Since the execution times of *TopK-SPLIT* are proportional to the number of substring edit distance computations, for each $G' \subseteq G$, we will use the expected number of such strings in D to decide the best $G' \subseteq G$. We represent the expected number of such strings in D by $Cost(G')$ defined below.

$$Cost(G') = D_1(G') + D_2(G'). \tag{5.2}$$

Estimation of $D_1(G')$: Assume that in every string in D, there always exists a substring obtained by performing edit operations randomly on each position in σ. Let β be the probability with which an edit operation is performed on a position in σ.

Let $E(\sigma, \rho)$ be the probability that at most ρ edit operations are performed among $|\sigma|$ positions in σ. Assuming edit operations occur independently and identically in every position of σ, $E(\sigma, \rho)$ can be computed by the Binominal distribution [88] as follows:

$$E(\sigma, \rho) = \sum_{n=0}^{\rho} \binom{|\sigma|}{n} \beta^n (1-\beta)^{|\sigma|-n}$$

We refer to the string containing a substring, to which σ is randomly transformed with at most $|G'|$ edit operations, as a *random match*. While we scan every string s in D, the probability that the string s is a random match is $E(\sigma, |G'|)$ as we defined above. We estimate $D_1(G')$ as the expected number of strings in D to examine until we find k random matches. Then, D_1 follows the negative binomial distribution $NB(k, 1-E(\sigma, |G'|))$ [90]. Since the expectation of random variable which follows the negative binomial distribution $NB(r, 1-p)$ is $(1-p) \cdot r/p$, the expected value of $D_1(G')$ is

$$D_1(G') = \frac{(1 - E(\sigma, |G'|)) \cdot k}{E(\sigma, |G'|)}. \tag{5.3}$$

Computation of $D_2(G')$: Remember that $D_2(G')$ is the number of strings that include a q-gram in G' among the rest of unseen strings after the condition in Lemma 5.1.10 is satisfied (i.e., we reach the critical point string). Assume that the strings with at least a q-gram in G' is distributed evenly in the database D. Let $L(G')$ be the number of strings with a q-gram of G' in D. Thus, the probability with which a string in D has a q-gram in G' is $L(G')/|D|$ where $|D|$ denotes the total number of strings in D. Then, after we find k strings whose substring edit distances are at most $|G'|$ (i.e., $\ell o(d_{sub}(D^{-}_{G'}))$), among the rest of strings in D (i.e., $|D| - D_1(G')$), we estimate $D_2(G')$ by computing the expected number of strings that contain at least

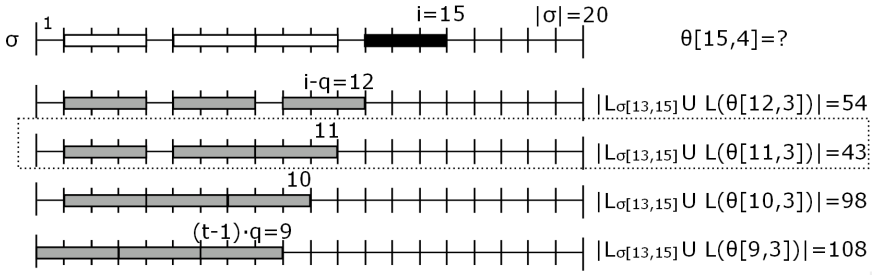


Figure 5.7: Computing $u[i,t]$ and $\theta[i,t]$

a q-gram in G' and it is

$$D_2(G') = \big(|D| - D_1(G')\big) \cdot \frac{L(G')}{|D|}. \tag{5.4}$$

Due to Corollary 5.1.9, we cannot select G' such that $|G'| > \lfloor |\sigma|/q \rfloor$ and thus we should select G' with the minimum $Cost(G')$ among all $G' \subseteq G$ whose sizes are in the range [1, $\lfloor |\sigma|/q \rfloor$]. However, note that $D_1(G')$ in Equation (5.3) depends on the size of G' only and with a fixed size of G', $D_2(G')$ in Equation (5.4) depends on $L(G')$ only. Thus, for each size of G', $Cost(G')$ becomes the minimum when $L(G')$ is the smallest. We refer to this algorithm for finding the best G' among all subsets of G as *BEST-G'* which is used in *TopK-SPLIT*.

We next propose the algorithm *SELECT-G'(ρ)* to find such G' when the size of G', denoted by ρ, is given as input. We can find G' with the minimum $Cost(G')$ among all $G' \subseteq G$ with $1 \leq |G'| \leq \lfloor |\sigma|/q \rfloor$ by invoke *SELECT-G'(ρ)* with $1 \leq \rho \leq \lfloor |\sigma|/q \rfloor$ and selecting the one with the minimum cost.

Selecting a good G' with the size $|G'| = \rho$: We use the following notations to describe our dynamic programming formulation to select G'.

- Let L_{g_i} be the set of string ids including the q-gram $g_i \in G$.

- For a q-gram set $Q \subseteq G$, let $L(Q) = \bigcup_{g_i \in Q} L_{g_i}$, which is the union of L_{g_i}s for all q-grams $g_i \in Q$, and we will use $|L(Q)|$ to represent the number of elements in $L(Q)$.

- Let $Q(i,t)$ be the set of all possible subsets Q of q-grams appearing in $\sigma[1,i]$ such that (1) $|Q|$=t, (2) the q-gram $\sigma[i{-}q{+}1,i]$ (i.e., the q-gram ending at the position i) always appears in Q and (3) all q-grams in Q are non-overlapping to each other.

- Let $\theta[i,t]$ denote the q-gram set Q in $Q(i,t)$ such that $|L(Q)|$ is the minimum among all q-gram sets in $Q(i,t)$.

- Let $u[i,t]$ represent $|L(\theta[i,t])|$.

Given the size ρ of G', we select Q with the minimum $|L(Q)|$ as G' among all subsets $Q \subseteq G$ consisting of ρ non-overlapping q-grams in σ. In every possible such Q, the ending position of the rightmost q-gram $\sigma[i{-}q{+}1,i]$ can be located in the positions i of σ with $\rho{\cdot}q{\leq}i{\leq}|\sigma|$. Thus, for every substring $\sigma[1,i]$ with $\rho{\cdot}q{\leq}i{\leq}|\sigma|$, we will enumerate the best ρ non-overlapping q-gram set Q which not only contains $\sigma[i{-}q{+}1,i]$ but also has the minimum $|L(Q)|$. The reason why we do not consider every i which is smaller than ($\rho{\cdot}q$) is because the substring $\sigma[1,i]$ with $i < \rho{\cdot}q$ cannot have ρ non-overlapping q-grams. Since we use $\theta[i,\rho]$ to store the best Q with ρ non-overlapping q-grams which not only contains $\sigma[i{-}q{+}1,i]$ but also has the minimum $|L(Q)|$, we can find G' by selecting the $\theta[i,\rho]$ with the smallest $|L(\theta[i,\rho])|$ among $\theta[i,\rho]$s with $\rho \cdot q{\leq}i{\leq}|\sigma|$. In other words, if we compute $\theta[i,t]$s for every $1 \leq t \leq \lfloor|\sigma|/q\rfloor$ and $t \cdot q \leq i \leq |\sigma|$, we can select the q-gram set $\theta[i^*,\rho]$ as G' where i^* is

chosen as follows:

$$i^* = \arg \min_{t \cdot q \leq i \leq |\sigma|} \{u[i, \rho]\} = \arg \min_{t \cdot q \leq i \leq |\sigma|} \{|L(\theta[i, \rho])|\}. \quad (5.5)$$

We next present a dynamic programming for computing $\theta[i, t]$ for $t = 1, \ldots, \lfloor |\sigma|/q \rfloor$ and $i = t \cdot q, \ldots, |\sigma|$.

Dynamic programming formulation: Since $\theta[i, t]$ contains t number of non-overlapping q-grams in the substring $\sigma[1, i]$ including $\sigma[i-q+1, i]$, we consider $\theta[j, t-1] \cup \{\sigma[i-q+1, i]\}$ with $1 \leq j \leq (i-q)$ to compute $\theta[i, t]$. However, $\theta[j, t-1]$ does not exist for $j = 1, \ldots, (t-1) \cdot q$-1 because it is impossible to have $(t-1)$ non-overlapping q-grams in the substring $\sigma[1, j]$. Thus, we enumerate $\theta[j, t-1] \cup \{\sigma[i-q+1, i]\}$ for $(t-1) \cdot q \leq j \leq (i-q)$ only and select $\theta[j, t-1] \cup \{\sigma[i-q+1, i]\}$ with the smallest $|L(\theta[j, t-1]) \cup L_{\sigma[i-q+1,i]}|$ as $\theta[i, t]$. Thus, our recursive definition for $\theta[i, t]$ is

$$\theta[i, t] = \theta[j^*, t-1] \cup \{\sigma[i-q+1, i]\} \quad (5.6)$$

where

$$j^* = \arg \min_{(t-1) \cdot q \leq j \leq i-q} \left\{ \left| L_{\sigma[i-q+1,i]} \cup L(\theta[j, t-1]) \right| \right\}. \quad (5.7)$$

We can also simply compute $u[i, t] = |L(\theta[i, t])|$.

In Figure 5.7, we show an example to illustrate how we compute $\theta[15, 4]$. We enumerate $|L_{\sigma[13,15]} \cup \theta[j, 3]|$ with $9 \leq j \leq 12$ and select $\{\sigma[13, 15]\} \cup \theta[11, 15]$ as $\theta[15, 4]$ which minimizes $|L(\theta[15, 4])|$.

If we compute $\theta[i, t]$ for every t from 1 to $\lfloor |\sigma|/q \rfloor$ and every i from $t \cdot q$ to $|\sigma|$ with the above dynamic programming algorithm, we can choose G' with a size ρ by Equation (5.5). We refer to this algorithm which finds G' with a given size ρ as *SELECT-G'(ρ)*.

Note that *SELECT-G'(ρ)* cannot find the optimal q-gram set G' because the optimal substructure property of our dynamic programming formulation is not satisfied. Suppose $\{\sigma[i-q+1, i]\} \cup Q$ is set to $\theta[i, t]$ in Equation (5.6) where Q is the optimal set with $(t-1)$ non-overlapping q-grams for $\sigma[1, j^*]$. Let Q' be a non-optimal q-gram set with $(t-1)$ non-overlapping q-grams in $\sigma[1, j^*]$ including the q-gram ending at the position j^*. Even though Q' is not an optimal q-gram set for $\sigma[1, j^*]$, if $L_{\sigma[i-q+1,i]}$ and $L(Q')$ share many common string ids so that $|L_{\sigma[i-q+1,i]} \cup L(Q')|$ becomes smaller than $|L_{\sigma[i-q+1,i]} \cup L(Q)|$, we have $|L(Q')| > |L(Q)|$ and thus the computed $\theta[i, t]$ is not an optimal q-gram set for $\sigma[1, i]$. Thus, *SELECT-G'(ρ)* finds an approximate q-gram set for G'. However, our performance study confirms that *SELECT-G'(ρ)* obtains good G's close to the optimal q-gram sets.

When *SELECT-G'(ρ)* is executed, we cannot know the actual size of $L(\theta[i, t])$ which is the union of the sets of string ids whose strings contain a q-gram in $\theta[i, t]$. We will use the MinHash technique presented in Section 3.3 to estimate the sizes of unions in the algorithm *SELECT-G'(ρ)*.

Assume that for every q-gram g_i appearing in D, there is a MinHash signature of the set of string ids in which the q-gram g_i occurs. In *SELECT-G'(ρ)*, we maintain the MinHash signature of $L(\theta[i, t])$ to compute $L_{\sigma[i-q+1,i]} \cup L(\theta[j, t-1])$ in Equation (5.7) with constant time.

Time complexity: In *SELECT-G'(ρ)*, we have to compute $\theta[i, t]$ and $u[i, t]$ for every $1 \leq t \leq \lfloor |\sigma|/q \rfloor$ and every $t \cdot q \leq i \leq |\sigma|$. For computing each $\theta[i, t]$, we need $O(|\sigma|)$ number of computations. For each $u[i, t]$, it takes constant time since $u[i, t]$ is simply $|L(\theta[i, t])|$. Thus, the time complexity of *SELECT-G'(ρ)* is $O(|\sigma|^3/q)$.

Example 5.3.1 Consider the query string σ='Jacksonv' with $|\sigma| = 8$. Suppose that

g_1 \ g_2	-	Jac	ack	cks
Jac	100	-	-	-
ack	32	-	-	-
cks	16	-	-	-
kso	10	110	-	-
son	120	130	125	-
onv	40	120	43	50

(a) $|L_{g_1} \cup L_{g_2}|$ (=$|L(\{g_1,g_2\})|$)

	t=1		t=2	
i	θ[i,1]	u[i,1]	θ[i,2]	u[i,2]
3	{Jac}	100	-	-
4	{ack}	32	-	-
5	{cks}	16	-	-
6	{kso}	10	{Jac,kso}	110
7	{son}	120	{ack,son}	125
8	{onv}	40	{ack,onv}	43

(b) θ[i,t] and u[i,t]

Figure 5.8: Computations in *SELECT-G'(ρ)*

we should select the non-overlapping 3-gram set G' with size 2. For all possible non-overlapping 3-gram sets Q whose sizes are 1 or 2, $|L(Q)|$s are shown in Figure 5.8(a). We also show all $\theta[i,t]$s and $u[i,t]$s computed by *SELECT-G'(ρ)* in Figure 5.8(b).

Let us assume that we want to compute $\theta[8,2]$ that always includes the 3-gram $\sigma[6,8]$='onv'. We enumerate the following three cases

- $|L_{onv} \cup \theta[5,1]| = 50$ (=$|L_{onv} \cup L_{cks}|$)
- $|L_{onv} \cup \theta[4,1]| = 43$ (=$|L_{onv} \cup L_{ack}|$)
- $|L_{onv} \cup \theta[3,1]| = 120$ (=$|L_{onv} \cup L_{Jac}|$)

due to Equation (5.7) and select $\theta[8,2]$ = {'ack','onv'} with $u[8,2] = 43$. Finally, to select the best G' with size 2 by Equation (5.5), we choose $\theta[8,2]$ = {'ack', 'onv'} for G' since $u[8,2]$ is the minimum among $u[6,2]$, $u[7,2]$ and $u[8,2]$. ∎

5.4 Experiments

We empirically compared the performance of our proposed algorithms. All experiments reported in this section were performed on the machines with Intel(R) Core(TM)2 Duo CPU 2.20GHz of processor and 2GB of main memory running Linux operating systems. All algorithms were implemented using C++ and compiled with GCC Compiler of version 4.1.3.

5.4.1 Implemented Algorithms

We implemented the following algorithms for our performance study.

- **TopK-NAIVE:** This represents the brute-force algorithm which computes the substring edit distance with every string in D to find the top-k approximate substring matches.

- **TopK-LB:** It is the implementation of *TopK-LB* which computes $d_{sub}(s, \sigma)$ only for the strings s in D whose lower bound $\ell o(d_{sub}(s, \sigma))$ is smaller than the k-th smallest substring edit distance found so far.

- **TopK-SPLIT:** This is the algorithm which improves *TopK-LB* by ignoring the computation of the substring edit distances of the strings which do not share any common q-gram with G' after the critical point string.

- **TopK-NGPP:** This is the modified version of *NGPP* in [85] to obtain the top-k approximate substring matches. The algorithm *NGPP* is the state-of-the-art algorithm to find all substrings of each string in D whose edit distances to a query string are at most a given maximum threshold τ. To adapt *NGPP* to top-k approximate substring matching, we first read the first k strings in D and set the k-th smallest one among their substring edit distances as the initial threshold τ. Then, as examining every string in D, if the k-th smallest substring edit distance becomes smaller than τ, we update the threshold τ to the k-th smallest substring edit distance.

Note that we used our own buffer management for scanning strings and accessing inverted indexes in all our implementations without utilizing OS buffers to see

the buffering effects for the tested algorithms more carefully. We report the native execution times (i.e., wall clock times) of the tested algorithms in this section.

5.4.2 Data Sets

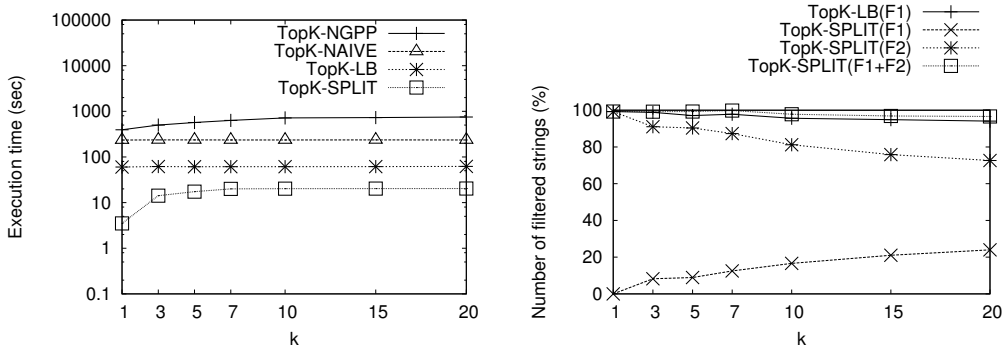
To study the performance our proposed algorithms, we utilize the following two real-life data sets.

- **DBLP:** To evaluate our algorithms with short strings, we used DBLP titles collected from [51]. However, since the original data is small, we increased its size by duplicating the original data 5 times. While we duplicate each string, we randomly performed an edit operation such as insert, delete and substitute on each position of the string with the probability of 0.1. The size of generated data set is 635 MB with 13,966,030 strings where the average size of strings is 48 bytes.

- **Wikipedia:** This is the data set consisting of 106,185 web pages obtained from [89] for the long string data set. The data size is 1.1 GB and the average size is 11,027 bytes.

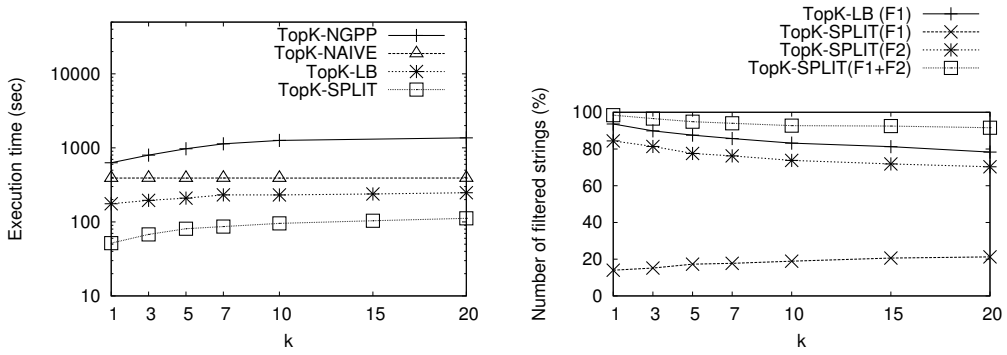
5.4.3 Queries Used

For DBLP, we randomly selected between 1 and 4 adjacent words appearing in DBLP titles to generate test queries and the range of query lengths is from 5 to 25. For Wikipedia, we sampled 50 entities from the name entity data collected for Name Entity Recognition in [20] as test queries and the query lengths are between 5 to 26.



(a) Execution times (DBLP)

(b) Filtering effects (DBLP)



(c) Execution times (Wikipedia)

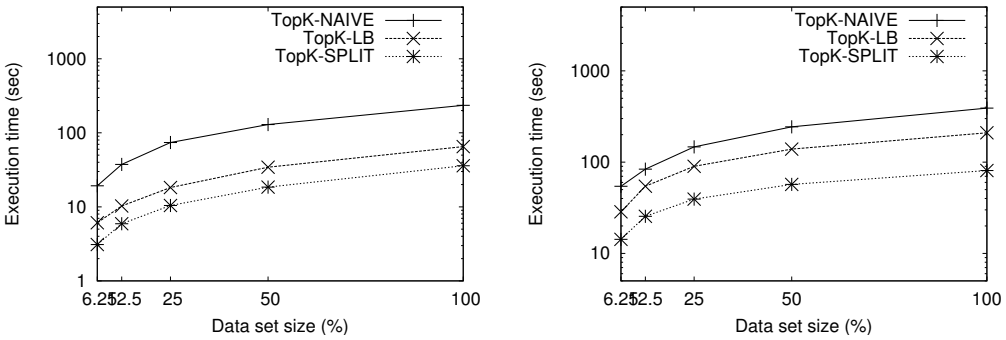
(d) Filtering effects (Wikipedia)

Figure 5.9: Varying k using DBLP and Wikipedia

5.4.4 Performance Results

We evaluated the proposed algorithms in terms of execution times. We measured the performance of the algorithms for both DBLP and Wikipedia with varying k and the number of strings n. Furthermore, we varied the length of q-grams q, MinHash signature size ℓ, the editing probability β (introduced in Section 5.3) and buffer size B used. The default parameters are: k=5, q=3, ℓ=50, β=0.85 and B=512MB.

Varying k: We first report the execution times, the number of filtered strings and the percentage of strings read from D with varying k from 1 to 20 in Figure 5.9(a)–(d).



(a) Varying n (Wikipedia) (b) Varying n (DBLP)

Figure 5.10: Varying n

(1) Execution times: We show the execution times for DBLP and Wikipedia in Figure 5.9(a) and Figure 5.9(c) respectively. The log scale was used on the y-axises. In both data, *TopK-NGPP* shows even worse performance than our naive algorithm *TopK-NAIVE* since *TopK-NGPP* has to enumerate all substrings of every string in D to compute the substring edit distances. As k is increased, the execution times for *TopK-LB* and *TopK-SPLIT* grow gradually. Since the k-th smallest substring edit distance so far becomes larger with growing k, less number of strings are skipped for computing substring edit distances. The graphs confirm that for every range of k, *TopK-SPLIT* shows the best performance. With DBLP, when k=1, *TopK-SPLIT* is 76 times faster than *TopK-NAIVE*. Furthermore, when k=20, *TopK-SPLIT* is 11 times faster than *TopK-NAIVE*. With Wikipedia, *TopK-SPLIT* is faster than *TopK-NAIVE* by 17 times when k=1 and 5.6 times when k=20.

(2) Filtering effects: To show the effectiveness of our filtering methods, we plotted the percentage of strings skipped for computing their substring edit distances in Figure 5.9(b) and Figure 5.9(d) with DBLP and Wikipedia respectively. Note that

TopK-LB(F1) in the graphs represents the percentage of strings filtered with *TopK-LB* because the lower bounds of their substring edit distances are at least the k-th smallest distance so far. We also use *TopK-SPLIT(F1)* similarly in the graphs. *TopK-SPLIT(F2)* in the graphs represents the percentage of strings filtered by *TopK-SPLIT* after the critical point string since they do not have any common q-gram with G' (i.e., they belong to $D^{-}_{G'}$).

With increasing k, both *TopK-LB(F1)* and *TopK-SPLIT(F1)* decrease gradually since the k-th smallest distance so far also becomes larger together with k. However, with growing k, *TopK-SPLIT(F2)* increases on the contrary. This is because $D_1(G')$ in the cost formula of Equation (5.2) is proportional to k and thus, if k is large, *BEST-G'* selects G' which decreases $D_2(G')$ much more. To decrease $D_2(G')$, *BEST-G'* should choose a small G' and with a small G', the number of strings in $D^{-}_{G'}$ which we encounter after the critical point string increases.

Varying n: With each of DBLP and Wikipedia, we selected strings from the original data set to produce smaller data with varying the sampling rate from 6.25% to 100%. With varying the size of data, we plotted the execution times in Figure 5.10(a) for DBLP and in Figure 5.10(b) for Wikipedia respectively. The graphs show that *TopK-SPLIT* is the fastest in every range of data sizes. Furthermore, as the data size increases, the relative speedup of *TopK-SPLIT* to *TopK-NAIVE* also improves from 4.4 times to 8.3 times for Wikipedia. With DBLP, the speedup increases from 9.3 times to 24.5 times. Thus, we conclude that *TopK-SPLIT* is scalable for large data.

Varying q: Figure 5.11 represents the graph of execution time as the length q of the q-grams used is varied from 3 to 5. Since *TopK-NAIVE* is not affected by the length of q-grams, its execution times are always constant. *TopK-LB* and *TopK-SPLIT*

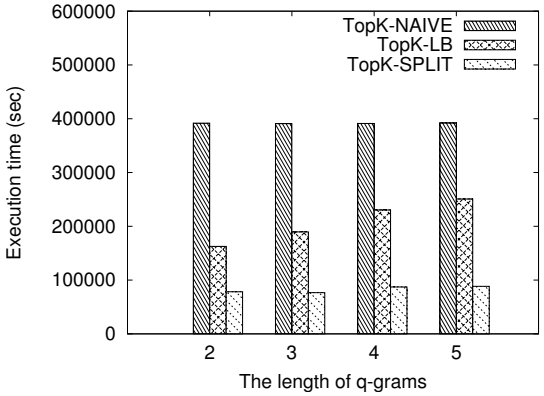
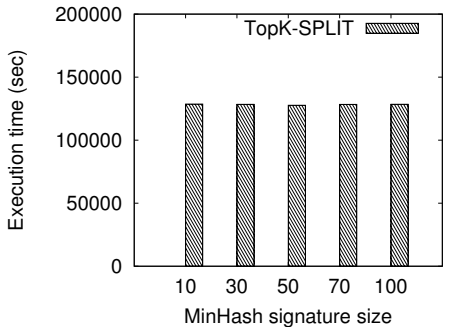


Figure 5.11: Varying q

show the best performances when q=2. This is because, with a large q, the lower bound $\lceil(|\sigma| - q + 1 - n/q\rceil$ in Lemma 5.1.6 decreases and thus, less strings are skipped for computing substring edit distances. Furthermore, since the maximum of $\ell o(d_{sub}(D^{-}_{G'}, \sigma))$ due to Corollary 5.1.9 which is $\lfloor|\sigma|/q\rfloor$ also becomes smaller with a larger q, the critical point string where we meet the k-th string whose substring edit distance is at most $\ell o(d_{sub}(D^{-}_{G'}, \sigma))$ appears later. However, *TopK-SPLIT* shows the best performance when q=3 because the posting lists become very large with 2-grams.

Varying ℓ: We varied the MinHash signature size ℓ from 10 to 100 and plotted the running times of *TopK-SPLIT* in Figure 5.12(a). Since the other algorithms do not estimate the computational cost to select G', they are not affected by the MinHash signature size. The graph shows that the performances of both algorithms are the worst when ℓ=10 and do not degrade that much with $\ell \leq 50$. Thus, we used ℓ=50 as the default value in all our experiments.

Varying β: To choose the proper value of β which represents the probability with



(a) Varying ℓ

β	TopK-SPlIT
0.95	127.512
0.9	127.323
0.85	127.662
0.8	136.249
0.75	145.223
0.7	150.532

(b) Varying β

Figure 5.12: Varying ℓ and β

which each character in query strings may be edited when the query strings appear in D (see Section 5.3), we show the execution times of *TopK-SPLIT* with varying β from 0.7 to 0.95 in Figure 5.12(b). The performance does not change much with $\beta > 0.85$ but gets worse rapidly with $\beta < 0.75$. Thus, we used 0.85 as the default value of β in our experiments.

Chapter 6

Top-k Approximate String Joins

To find similar documents based on the distributions of the words, the vector space model[71] has been widely used because of its conceptual simplicity[28, 68, 5]. In the vector space model, documents or query strings are represented as vectors in a multidimensional space, where each dimension corresponds to the frequency of a word. Thus, in the vector space model, the similarity between two documents is calculated using the distance between two vectors transformed from the two documents respectively.

To represent documents in the vector space model, many dimensionality reduction techniques have been studied since the number of dimensions in the vector space is generally very large. It includes principal component analysis[38], multidimensional scaling[80, 23], probabilistic latent semantic indexing[33, 34], latent Dirichlet allocation[12] and non-negative matrix factorization[46]. With the vectors of reduced dimensions obtained from text documents, called feature vector, the Euclidean distance is frequently used to compute the similarity between documents in many text classification and clustering algorithms[34, 33, 23]. Thus, to handle

the top-k approximate string join queries, we will focus on the problem of finding the top-k closest pairs of vectors in an Euclidean space.

The problem of finding the closest pairs using the Euclidean distance has been extensively studied in computational geometry. Given n number of d-dimensional vectors, a divide-and-conquer algorithm, which finds the top-1 closest pair of vectors with the time complexity $O(4^d \cdot n(\log n)^{d-1})$, was proposed in [9]. Later on, to find the top-k closest pairs, the algorithms with $O(20^d \cdot n \log n + k)$ time for L_∞-distance were introduced by Salowe in [70], and by Lenhof and Smid in [49] independently. However, these algorithms are exponential to the number of dimensions and thus they do not scale for high dimensional data. To speed up the top-k similarity join in an Euclidean space, the algorithms in [21] and [22] were proposed using spatial disk-based indexes such as R*-trees. However, they can handle two dimensional data only. Due to the curse of dimensionality[30], it is not easy to develop efficient top-k similarity join algorithm utilizing such spatial indexes.

In this chapter, given a set of feature vectors obtained from text data, we develop efficient similarity join algorithms finding the top-k close pairs of vectors in an Euclidean space. We first propose efficient divide-and-conquer and branch-and-bound algorithms, which prune the distance computations efficiently, called *TopK-D* and *TopK-T* respectively. To further improve those proposed algorithms, we next propose a partitioning based method for pruning distance computations, called *essential pair partitioning*. The partitioning distributes only a smaller subset of all pairs of vectors into partitions while we guarantee that the partitions always include the top-k closest pairs of the given data. Using the *essential pair partitioning*, we can reduce the number of distance computations effectively in our *TopK-D* and *TopK-T* algorithms.

p_1	<0.78, 0.4, 0.01>
p_2	<0.07, 0.21, 0.57>
p_3	<0.51, 0.11, 0.32>
p_4	<0.31, 0.79, 0.9>
p_5	<0.77, 0.42, 0.02>
p_6	<0.8, 0.39, 0.04>

(a) Feature vectors

	p_2	p_3	p_4	p_5	p_6
p_1	0.924	0.503	1.079	0.024	0.037
p_2		0.516	0.709	0.915	0.920
p_3			0.916	0.504	0.491
p_4				1.060	1.068
p_5					0.047

(b) The Euclidean distances of all pairs

Figure 6.1: An example of feature vectors

In the following, we define the notations used in this chapter and provide the problem definition of top-k approximate string join.

Problem definition: Let *fvector(s_i)* represent a function which generates and returns the d-dimensional feature vector p_i with a document s_i in D. Each feature vector p_i is represented as $\langle p_i(1),\ p_i(2), ..., p_i(d)\rangle$ where $p_i(j)$ denotes the j-th coordinate of p_i which is a real number. Let P=$\{p_1$,...,$p_n\}$ be the collection of d-dimensional vectors where each vector p_i is obtained by calling *fvector(s_i)* with the document $s_i \in D$. We consider the Euclidean distance only, but our work can still be applied to Minkowski distance L_γ [72]. The Euclidean distance $d_2(p_i, p_j)$ between the vectors $p_i = \langle p_i(1), p_i(2), ..., p_i(d)\rangle$ and $p_j = \langle p_j(1), p_j(2), ..., p_j(d)\rangle$ is defined as $d_2(p_i, p_j) = (\sum_{\ell=1}^{d} |p_i(\ell) - p_j(\ell)|^2)^{1/2}$.

The problem of top-k approximate string join is defined as follows:

Definition 6.0.1 Given a set of d-dimensional feature vectors $P = \{p_1$,..., $p_n\}$ with $p_i \in \mathbb{R}^d$, the problem of top-k approximate string join is to find the top-k closest pairs of vectors which consist of k distinct pairs of vectors, $TopP(k) = \{(p_{i_1},\ p_{j_1})$, ..., $(p_{i_k},\ p_{j_k})\}$ satisfying the following conditions:

- $d_2(p_{i_1}, p_{j_1}) \leq d_2(p_{i_2}, p_{j_2}) \leq \cdots \leq d_2(p_{i_k}, p_{j_k})$ holds.

- For every pair (p_{i_ℓ}, p_{j_ℓ}) $\in TopP(k)$, we have $i_\ell < j_\ell$ and p_{i_ℓ}, $p_{j_\ell} \in P$.
- For every pair (p_{i_ℓ}, p_{j_ℓ}) with $i_\ell < j_\ell$, (p_{i_ℓ}, p_{j_ℓ}) $\notin TopP(k)$ and p_{i_ℓ}, $p_{j_\ell} \in P$, we have $d_2(p_{i_k}, p_{j_k}) \leq d_2(p_{i_\ell}, p_{j_\ell})$.

Example 6.0.2 Assume that the feature vectors obtained from the text data are given as Figure 6.1(a). The Euclidean distance between every pair of vectors is also provided in Figure 6.1(b). The result of top-3 similarity join becomes $TopP(3)$={(1,5), (1,6), (5,6)}. ∎

We will use the top-k approximate string join and the top-k approximate join problems interchangeably in this chapter.

6.1 Top-k Approximate Join Algorithms

In this section, we propose two novel main-memory based algorithms based on divide-and-conquer and branch-and-bound. In the rest of this chapter, we will refer to the brute-force algorithm, which computes the distance of every possible pair of vectors with $O(d \cdot \log k \cdot n^2)$ time, as *TopK-B*.

6.1.1 TopK-D: Divide-and-Conquer Algorithm

We generalize the divided-and-conquer algorithm in [9] which finds the top-1 closest pair only. We refer to our generalized algorithm for finding the top-k closest pairs of vectors as *TopK-D*.

Given a data vectors D, we divide the vectors in D by a hyperplane and find the top-k closest pairs on each side recursively. To compute the distances of the pairs crossing the hyperplane, the algorithm in [9] invokes recursions with next splitting

Function TopK-D (m, k, X_m, ..., X_d)
begin
1. **if** $|X_m| \leq 2$ **then**
2. **if** $|X_m| = 2$ **then return** $\{\langle X_m[1], X_m[2]\rangle\}$;
3. **else return** $\{\}$;
4. $\ell = X_m[|X_m|/2](m)$;
5. $X_j^R = X_j^L =$ empty arrays for $m \leq j \leq d$;
6. **for** $i = 1$ to $|X_m|$ **do**
7. **for** $j = m$ to d **do**
8. **if** $X_j[i](m) < \ell$ **then** Append $X_j[i]$ to X_j^L;
9. **else** Append $X_j[i]$ to X_j^R;
10. H^L = TopK-D (m, k, X_m^L, ..., X_d^L);
11. H^R = TopK-D (m, k, X_m^R, ..., X_d^R);
12. $H = H^L \cup H^R$;
13. **if** $|H| < k$ **then** $\delta = \infty$;
14. **else** $\delta = d(H[k])$;
15. **if** $m + 1 = d$ **then** {
16. H = empty max-heap of pairs ordered by distance;
17. **for** i=1 to $|X_{m+1}^R|$ **do** {
18. Remove $p \in buf$ if $|p(m+1) - X_{m+1}^R[i](m+1)| > \delta$;
19. **for** j=1 to $|buf|$ **do** {
20. **if** $(buf[j](m) - \ell) \cdot (X_{m+1}^R[i](m) - \ell) < 0$ **then**
21. Insert $\langle buf[j], X_{m+1}^R[i]\rangle$ into H;
22. **if** $|H| > k$ **then** DeleteMax(H);
23. }
24. Insert $X_{m+1}^R[i]$ to buf;
25. }
26. **return** H;
27. }
28. **for** $i = 1$ to $|X_m|$ **do**
29. **for** $j = m$+1 to d **do**
30. **if** ℓ-$\delta \leq X_j[i](m) \leq \ell$+$\delta$ **then** Append $X_j[i]$ to B_j;
31. **return** TopK-D($m + 1$, k, B_{m+1}, ..., B_d);
end

Figure 6.2: The TopK-D algorithm

dimensions until there remains only a single dimension not split yet so that we can consider only constant number of distance computations for each vector when we compute the distances of the pairs crossing the hyperplane. For computing the closest pair of d-dimensional vectors, it is shown in [78] that there can be $O(4^d)$ vectors whose every pair is not closer than the closest distance when we calculate the distances of the pairs crossing the hyperplane. So, when we find the top-k closest pairs, for each vector p, we can guarantee that there are $O(k + 4^d)$ vectors whose distances from p is at most the distance of the k-th closest pair.

The pseudocode of *TopK-D* is presented in Figure 6.2. The recursive function takes the dimension m for dividing vectors, k, and (d-m+1) arrays, X_m, ..., X_d as input. All arrays X_i with $m \leq i \leq d$ contain the same vectors and are sorted by the i-th coordinate of vectors. First, *TopK-D* checks the boundary condition whether $|X_m| \leq 2$. If the condition is true, it just returns the pair. Otherwise, the algorithm divides the input vectors into two partitions and invokes itself recursively for each partition. In lines 6–9, we split every X_i into two set with $m \leq i \leq d$, X_i^L and X_i^R of the same size, where the vectors in X_i^L are in the left side of the hyperplane $x_m = \ell$ and the vectors in X_i^R are in the right side $x_m = \ell$.

Let H and δ be the returned pairs from recursive calls for each side of $x_m = \ell$ and the distance of the k-th closest pair in H respectively. Now we are going to compute the distances of pairs crossing $x_m = \ell$ by recursively calling itself with the vectors whose distances of their m-th coordinates from ℓ are less than δ. Let B_i for $m \leq i \leq d$ be the vectors whose m-th coordinates are in the range of $[\ell - \delta, \ell + \delta]$ and B_i is sorted in monotonically increasing order of i-th coordinate. The recursive invocation takes m+1, k and B_i for $m + 1 \leq i \leq d$ as input. When $m + 1 = d$, we

compute the distance using sliding window along the array of the last coordinate.

Time complexity: For each recursion with n vectors and the dividing dimension m, the partitioning of (d-m+1) arrays, which are X_m, ..., X_d, is performed in $O(d \cdot n)$ time in lines 6–9 and the selection of the vectors in the range $[\ell - \delta, \ell + \delta]$ costs $O(d \cdot n)$ time in lines 28–30. Since each recursion invokes two recursive calls with $n/2$ vectors and the p-th dividing dimension as input, and invokes another recursive call with $O(n)$ vectors and p-1 as input, the recurrence for the running time is $T(n,p)$=$2T(n/2,p)$+$O(d \cdot n)$ +$T(n,p-1)$. In line 1, the boundary condition with $n = 2$ is $T(2,p) = \Theta(1)$, and in lines 15–27, since we get $p = 2$ from $m + 1 = d$, the boundary condition satisfying is $T(n,2) = O(\log k \cdot (k + 4^d) \cdot n)$. With the recurrence and boundary conditions, we obtain $T(n)$=$O(\log k \cdot (k + 4^d) \cdot n(\log n)^{d-1})$ where the solution for this recurrence was addressed in [60].

6.1.2 TopK-T: Branch-and-Bound Algorithm

In this section, we propose another top-k approximate join algorithm *TopK-T*, which uses pruning with the lower bound for the distances of unseen pairs yet as the *Threshold Algorithm* in [26] developed for top-k query evaluation problem does.

Threshold Algorithm (TA): Given a set of objects with d attributes and an d-ary monotonic score function s, *TA* finds the k objects with the smallest scores by utilizing the lower bound T of scores for pruning. Let $o_i(\ell)$ represent ℓ-th attribute value of an object o_i. A d-ary scoring function s is defined to be monotonic if for every pair of d-attribute objects o_i and o_j with $o_i(\ell) \leq o_j(\ell)$ for every ℓ-th attribute, the condition of $s(o_i(1), \ldots, o_i(d)) \leq s(o_j(1), \ldots, o_j(d))$ holds.

For a given set of objects, *TA* builds a sorted list L_ℓ of the objects for each ℓ-th

p_1	$\langle$0.78, 0.4, 0.01$\rangle$
p_2	$\langle$0.07, 0.21, 0.57$\rangle$
p_3	$\langle$0.51, 0.11, 0.32$\rangle$
p_4	$\langle$0.31, 0.79, 0.9 $\rangle$
p_5	$\langle$0.77, 0.42, 0.02$\rangle$
p_6	$\langle$0.8, 0.39, 0.04$\rangle$

Figure 6.3: A set of data points D

attribute in the increasing order of the attribute values where each entry in L_ℓ consists of the ℓ-th attribute value and the pointer to its object. *TA* iteratively accesses each sorted list L_ℓ in a round robin fashion until there is no need to access L_ℓs any more. For every ℓ-th attribute, we maintain the value u_ℓ, which is lastly seen in the list L_ℓ during previous iterations, and the lower bound threshold T=$s(u_1, \ldots, u_d)$.

When an entry with the pointer to an object o_i is seen with an access to L_ℓ in each iteration, *TA* performs a random access to the object and computes the overall score of the object o_i. If the score is smaller than the k-th smallest score so far, we update the top-k list. We also update the value of u_ℓ with the ℓ-th attribute value of o_i and assign T as $s(u_1, \ldots, u_d)$ with the updated u_ℓ. Since the ℓ-th attribute values of unseen objects are at least u_ℓ with every ℓ-th attribute, the overall scores of unseen objects cannot be smaller than T due to the monotonic scoring function s. Thus, in each iteration, if the score of the object with the k-th smallest score is at most T, we know that unseen objects cannot have smaller scores than the k-th smallest score and thus we can stop the iterations without examining the rest of unseen objects any more.

We next present how we can utilize the TA algorithm to find the top-k closest pairs of vectors in a given data.

Direct use of *TA* with $O(dn^2)$ space: Assume that we have a set of d-dimensional vectors $D = \{p_1,...,p_n\}$. We can consider as if every pair of vectors p_i, $p_j \in D$ with $i < j$ corresponds to an object. Since Euclidean distance function $d(p_i, p_j)$ is a monotonic function, we can utilize *TA* to find the top-k closest pairs. For every ℓ-th coordinate, we build a list L_ℓ of every pair p_i, $p_j \in D$ satisfying $i < j$, where each entry in L_ℓ has the pointers to both p_i and p_j, and sort L_ℓ in the increasing order of the distances in the ℓ-th coordinates only (i.e., $|p_i(\ell) - p_j(\ell)|$). Note that we want to store distinct pairs only and thus we store every pair p_i, $p_j \in D$ with $i < j$. We maintain $u_\ell = |p_i(\ell) - p_j(\ell)|$ for every ℓ-th dimension where (p_i,p_j) is the most recently accessed element in L_ℓ. We also compute $T = (\sum_{\ell=1}^{d} u_\ell^2)^{1/2}$ as the lower bound threshold of the distances of the remaining unseen pairs. Then, we can compute the top-k closest pairs similar to the TA algorithm.

Example 6.1.1 Suppose that we have 3-dimensional vectors as shown in Figure 6.3 and we are interested with top-2 closest pairs. Assuming that each distinct pair of vectors corresponds to an object for *TA* and each object has 3 attributes, a_1, a_2 and a_3. The key used to sort the list L_i is the distance of the i-th coordinate between two vectors. For a given object o, we use the score function $s(o) = \left(o.a_1^2 + o.a_2^2 + o.a_3^2\right)^{1/2}$ which computes the Euclidean distance between a pair of vectors and is a monotonic function. The sorted list L_is for top-k closest pair problem is shown in Figure 6.4.

We initially let u_1=u_2=u_3=0 and thus the lower bound threshold T=0. We assume that our round robin fashion examines the sorted list in the order of L_1, L_2 and L_3 cyclically. At the first iteration of *TA*, we access the first entry ($\langle p_1,p_5\rangle$, 0.01) in L_1 which corresponds to the distance between the first coordinates of p_1 and p_5. By accessing the pointers to p_1 and p_5, we get the distance between p_1 and p_5 is $\sqrt{0.0006}$. Now u_1 becomes 0.01 and T becomes $\sqrt{u_1^2 + u_2^2 + u_3^2}$=$0.01$. We next access to L_2 and get ($\langle p_1,p_6\rangle$, 0.01) which represents that the distance between the second coordinates of vectors p_1 and p_6 is 0.01. The Euclidean distance between them is $\sqrt{0.0014}$ which becomes the second smallest distance. We let $u_2 = 0.01$ and

$pair$	a_1
$\langle p_1,p_5\rangle$	0.01
$\langle p_1,p_6\rangle$	0.02
$\langle p_5,p_6\rangle$	0.03
$\langle p_3,p_4\rangle$	0.20
$\langle p_2,p_4\rangle$	0.24
$\langle p_3,p_5\rangle$	0.26
$\langle p_1,p_3\rangle$	0.27
$\langle p_3,p_6\rangle$	0.29
$\langle p_2,p_3\rangle$	0.44
$\langle p_4,p_5\rangle$	0.46
$\langle p_1,p_4\rangle$	0.47
$\langle p_4,p_6\rangle$	0.49
$\langle p_2,p_5\rangle$	0.70
$\langle p_1,p_2\rangle$	0.71
$\langle p_2,p_6\rangle$	0.73

(a) L_1

$pair$	a_1
$\langle p_1,p_6\rangle$	0.01
$\langle p_1,p_5\rangle$	0.02
$\langle p_5,p_6\rangle$	0.03
$\langle p_2,p_3\rangle$	0.10
$\langle p_2,p_6\rangle$	0.18
$\langle p_1,p_2\rangle$	0.19
$\langle p_2,p_5\rangle$	0.21
$\langle p_3,p_6\rangle$	0.28
$\langle p_1,p_3\rangle$	0.29
$\langle p_3,p_5\rangle$	0.31
$\langle p_4,p_5\rangle$	0.37
$\langle p_1,p_4\rangle$	0.39
$\langle p_4,p_6\rangle$	0.40
$\langle p_2,p_4\rangle$	0.58
$\langle p_3,p_4\rangle$	0.68

(b) L_2

$pair$	a_1
$\langle p_1,p_5\rangle$	0.01
$\langle p_5,p_6\rangle$	0.02
$\langle p_1,p_6\rangle$	0.03
$\langle p_2,p_3\rangle$	0.25
$\langle p_3,p_6\rangle$	0.28
$\langle p_3,p_5\rangle$	0.30
$\langle p_1,p_3\rangle$	0.31
$\langle p_2,p_4\rangle$	0.33
$\langle p_2,p_6\rangle$	0.53
$\langle p_2,p_5\rangle$	0.55
$\langle p_1,p_2\rangle$	0.56
$\langle p_3,p_4\rangle$	0.58
$\langle p_4,p_6\rangle$	0.86
$\langle p_4,p_5\rangle$	0.88
$\langle p_1,p_4\rangle$	0.89

(c) L_3

Figure 6.4: An example of the sorted list L_is for direct use of TA algorithm

$T = \sqrt{0.01^2 + 0.01^2 + 0} = \sqrt{0.0002}$. Since the second smallest distance $\sqrt{0.0014}$ is less than T, we continue to access L_3 in next iteration. Repeating above steps, when we reach to the third entry of L_1 with the two vectors p_5 and p_6, we get $u_1 = 0.03$, $u_2 = 0.02$ and $u_3 = 0.02$, and thus $T = \sqrt{u_1^2 + u_2^2 + u_3^2} = \sqrt{0.0017}$. Since the second smallest distance $\sqrt{0.0014}$ so far is smaller than T, we stop and obtain the top-2 closest pairs $\langle p_1,p_5\rangle$ and $\langle p_1,p_6\rangle$. ∎

However, it is not practical to use TA algorithm directly for finding top-k closest pair with $O(dn^2)$ space for large data, we next propose the algorithm *TopK-T* with $O(d \cdot n)$ space instead.

TopK-T: We now present our efficient branch-and-bound algorithm *TopK-T*. Instead of maintaining L_ℓs, we utilize the min-heaps Q_ℓs with $\Theta(n)$ space so that we can

Function TopK-T(D, k)
begin
1. TOP_k = empty max-heap;
2. **for** ℓ = 1 to d **do** {
3. S_ℓ = array of every vector p in D sorted by p_i;
4. Q_ℓ = min-heap of ($|S_\ell[i](\ell)$-$S_\ell[i+1](\ell)|, i, i+1$), $\forall 1 \leq i \leq |D|$-1;
5. u_ℓ = 0;
6. }
7. ℓ = 1, T = 0;
8. **while** ($|TOP_k| < k$ **or** GetMaxDistance(TOP_k) $> T$) **do**
9. **if** $|Q_\ell|$ = 0 **then** exit loop;
10. **else** {
11. $(dist_\ell, p, q)$ = PopMin (Q_ℓ);
12. $u_\ell = dist_\ell$;
13. **if** $y \leq |D| - 1$ **then**
14. Insert $(|S_\ell[p](\ell) - S_\ell[q+1](\ell)|, p, q+1)$ into Q_ℓ;
15. Insert $(dist(S_\ell[p], S_\ell[q]), \{p, q\})$ into TOP_k;
16. **if** $|TOP_k| > k$ **then** DeleteMax(TOP_k);
17. $T = (\sum_{i=1}^{d} u_i^2)^{1/2}$;
18. $\ell = (\ell \bmod d) + 1$;
19. }
20. **return** pairs in TOP_k;
end

Figure 6.5: The TopK-T algorithm

retrieve the next closest pair of vectors one by one exactly in the same order of accessing L_ℓs. Thus, we will store only the next closest pair for every vector into the min-heap Q_ℓ. As we extract and delete the closest pair with the vector p_i from Q_ℓ, we insert the next closest pair with p_i again.

For every ℓ-th dimension, we first build the sorted list S_ℓ with all vectors in D in the increasing order of the ℓ-th coordinate. Let $S_\ell[i]$ be the i-th vector in the list S_ℓ for the ℓ-th coordinate and let $S_\ell[i](\ell)$ denote the ℓ-th coordinate of $S_\ell[i]$. When we

Idx	$p_i(1)$	Pid
1	0.07	p_2
2	0.31	p_4
3	0.51	p_3
4	0.77	p_5
5	0.78	p_1
6	0.80	p_6

(a) S_1

Idx	$p_i(2)$	Pid
1	0.11	p_3
2	0.21	p_2
3	0.39	p_6
4	0.40	p_1
5	0.42	p_5
6	0.79	p_4

(b) S_2

Idx	$p_i(3)$	Pid
1	0.01	p_1
2	0.02	p_5
3	0.04	p_6
4	0.32	p_3
5	0.57	p_2
6	0.90	p_4

(c) S_3

Figure 6.6: Sorted arrays for each coordinate

initially build a min-heap Q_ℓ, for every vector $S_\ell[i]$ with $1 \leq i \leq n-1$, we insert ($|S_\ell[i](\ell)$-$S_\ell[i+1](\ell)|$, i, i+1) into Q_ℓ where $|S_\ell[i](\ell)$-$S_\ell[i+1](\ell)|$ is the key used in the min-heaps. Note that $S_\ell[i+1]$ is the closest neighbor of the vector $S_\ell[i]$ according to the distance of the i-th coordinate only among the distinct pairs of vectors with the vector $S_\ell[i]$ due to the similar reason of enforcing the condition $i < j$ for every pair $p_i, p_j \in D$ stored in L_ℓ.

When we remove the top element $(dist, p, q)$ from a min-heap Q_ℓ in each iteration, the pair of $S_\ell[p]$ and $S_\ell[q]$ is used as if it is the next closest pair in L_ℓ based on the distance of the ℓ-th coordinate only. Then, $S_\ell[p]$ is inserted again as ($|S_\ell[p](\ell)$-$S_\ell[q+1](\ell)|, p, q$+1) into the min-heap Q_ℓ since $S_\ell[q+1]$ is the next closest vector of $S_\ell[p]$ appearing after $S_\ell[q]$.

The time complexity of *TopK-T* is $O(d^2n^2+dn^2 \log k+dn^2 \log n)$ which is worse than that of the brute-force algorithm *TopK-B* (i.e., $O(d \cdot n^2 \cdot \log k)$). However, we found by experiments that *TopK-T* performs better in practice compared to both *TopK-B* and *TopK-D* due to effective pruning.

The pseudocode of *TopK-T* is presented in Figure 6.5. *TopK-T* begins by initial-

izing an empty max-heap TOP_k to keep the top-k closest pairs. For each ℓ-th coordinate with $1 \leq \ell \leq d$, we first make a sorted array S_ℓ with every vector in D by the ℓ-th coordinate values and build a min-heap Q_ℓ by inserting the tuples $(|S_\ell[i](\ell) - S_\ell[i+1](\ell)|,\ i,\ i+1)$ for every i with $1 \leq i \leq n-1$. The while-loop iterates until there are at least k pairs in TOP_k and the distance of the k-th closest pair in TOP_k is at most the lower bound distance T of unseen pairs. The loop also stops when there remains no more entries in Q_ℓ. We get the distance of the k-th closest pair in TOP_k by invoking *GetMaxDistance(TOP_k)*. In each iteration of the while-loop, we first extract and delete the top tuple $(|S_\ell[p](\ell) - S_\ell[q](\ell)|,\ p,\ q)$ from Q_ℓ by calling *PopMin(Q_ℓ)*, update u_ℓ with $|S_\ell[p](\ell) - S_\ell[q](\ell)|$, and update T with u_ℓ. Then, since the vector which has the next closest ℓ-th coordinate from $S_\ell[p](\ell)$ is $S_\ell[q+1]$, we insert the tuple $(|S_\ell[p](\ell) - S_\ell[q+1](\ell)|,\ p,\ q+1)$ into Q_ℓ as long as the condition $q+1 \leq n$ is satisfied. If we exit the while-loop, we answer for the top-k closest pairs with the pairs in TOP_k.

Example 6.1.2 Consider the set of vectors D in Figure 6.3. Assume that we are interested in the top-2 closest pairs of vectors in D with Euclidean distance. We first generate the sorted lists S_1, S_2 and S_3 as shown in Figure 6.6. Initially, for each vector p in D, Q_i contains the difference of the i-th coordinate with the next vector of p in the array S_i. Then, Q_1 has {(0.01, 4, 5), (0.02, 5, 6), (0.2, 2, 3), (0.24, 1, 2), (0.26, 3, 4)}. For example, Q_1 contains (0.01, 4, 5) because the array index of p_5 is 4 in S_1 and the next vector of p_5 is p_1 whose array index is 5 in S_1. The first access to Q_1 returns a pair (p_5, p_1) where the difference between the first coordinates of the vectors is 0.01. With computing the distance of (p_5, p_1), which is 0.02, we insert the pair into TOP_k. Instead of (p_5, p_1) that is just extracted from Q_1, we add (0.03,4,6) which is the pair of vectors (p_5, p_6) into Q_1 since p_6 is listed in the next of p_1 in S_1. After the first access to Q_1, T becomes 0.01. Next access to Q_2 returns (0.01,3,4) for the pair (p_6, p_1) whose distance is $\sqrt{0.0014}$. The pair is inserted into TOP_k and

for the pair (p_6, p_5), (0.01,3,5) is inserted into Q_2 since p_5 is listed in the next of p_1. Since there are 2 pairs in TOP_k and the second closest pair distance, $\sqrt{0.0014}$, is larger than $T = \sqrt{0.0002}$, we continue the while-loop. For the next access, we return to Q_1 and similarly perform each step of the iteration. At the 7-th access, we get the pair (p_5, p_6) from Q_1 and T becomes $\sqrt{0.0017}$. The current second closest pair in TOP_k is (p_1, p_6) with the distance $\sqrt{0.0014}$. Since T gets larger than the distance of the second closest pair in TOP_k, we stop the while-loop. Finally, the top-2 closest pairs become (p_1, p_5) and (p_1, p_6). ∎

Time complexity: Each iteration takes $O(d+\log k+\log n)$ time for computing distance for a new pair and updating Q_ℓ. Since the total size of L_ℓs is $O(dn^2)$ and we may access all elements L_ℓs, the time complexity of *TopK-T* is $O(d^2n^2+dn^2 \log k+ dn^2 \log n)$.

6.2 Essential Pair Partitioning

In this section, we propose the essential pair partitioning which distributes only a subset of all pairs into the partitions while we guarantee that the partitions include the top-k closest pairs of the give data.

6.2.1 Safe Bucket Assignments

Even if we distribute only a subset of all pairs containing the top-k closest pairs into the partitions and find the top-k closest pairs in each partition only, we can always find the top-k closest pairs. Interestingly, if we know an upper bound τ of the distance of the k-th closest pair in the data, we can avoid distributing the pairs, whose distances are larger than τ, into the partitions. To reduce the number of pairs distributed, we first perform *space partitioning* in every dimension and distribute all

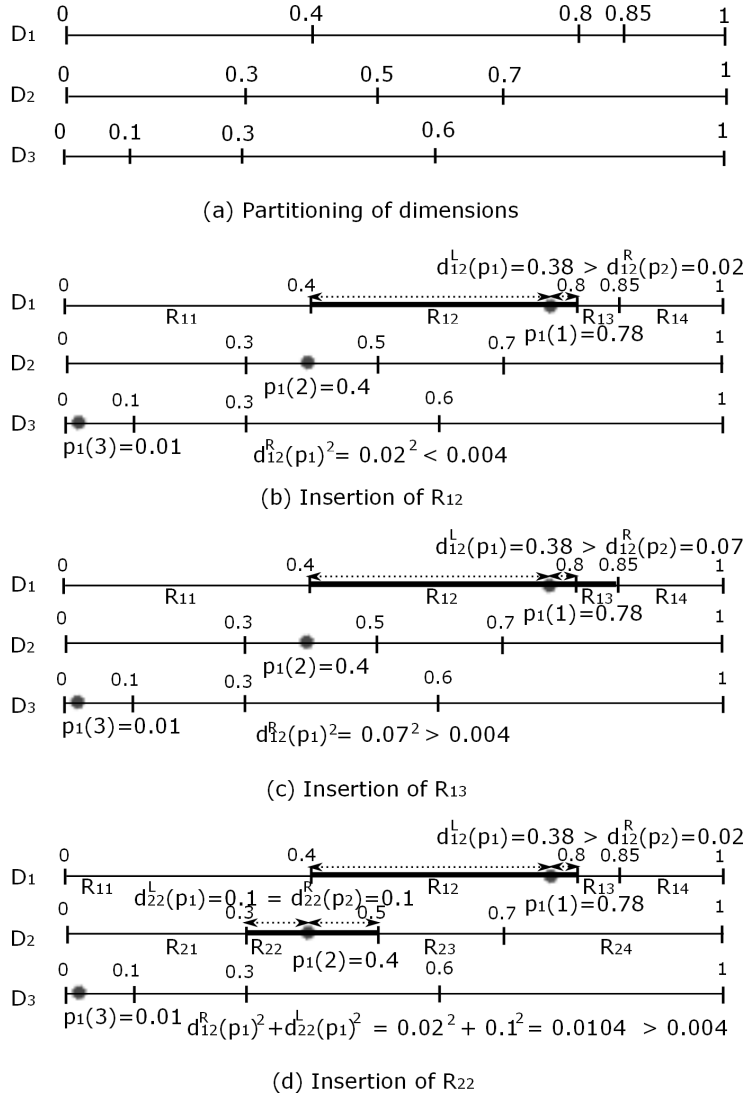


Figure 6.7: An example of essential pair partitioning

points into some buckets generated by space partitioning. To get a reasonable value of τ, we use a sampling method which will be discussed later.

Assume that our space partitioning splits the space in each dimension i into n_i distinct ranges R_{ij}s where R_{ij} is the j-th range of the dimension i and is represented

by $(s_{ij}, e_{ij}]$ with $1 < j \leq n_i$ and by $[s_{i1}, e_{i1}]$ with $j = 1$. We associate a bucket B_{ij} to each R_{ij} which will be used to assign the points. An example of partitioning every dimension into 4 ranges for 3 dimensional data (i.e., $n_1 = n_2 = n_3 = 4$) is shown in Figure 6.7(a).

Our goal of distributing the points into buckets is to make sure that every pair within distance τ appears in at least a single bucket while every pair with a distance larger than τ has high chances of not appearing in any bucket. Then, we can reduce the number of distance computations since we will compute the distances of the pairs in each bucket only to obtain the top-k closest pairs.

With a given space partitioning, we enforce that each point is inserted into the buckets in a given specific order of dimensions provided. Without loss of generality, we assume that the dimensions are ordered as $D_1, \ldots, D_d$. For each point p, we first insert it into the bucket with the range R_{1j}, whose range contains the first coordinate of p. Note that every point should be inserted at least into a single bucket in the first dimension. Then, we may insert it to other buckets, not inserted previously, in the same dimension or move to a bucket in the next dimension, but we cannot skip any dimension without an insertion in the given order of dimensions. If we select the buckets for every point without the same fixed order of dimensions, there may exists a pair of points whose distance is smaller than the threshold τ without any common bucket.

Furthermore, if a point p is inserted to the buckets in the i-th dimension, it should be inserted first to the single bucket with the range R_{ij} which contains p's value of the i-th dimension, and then be inserted to one of neighboring buckets of the previously selected buckets so that the selected ranges in each dimension form

a chain with contiguous ranges in the dimension. We refer to the insertion of the points in this fashion as the *safe bucket assignment*.

The problem with the safe bucket assignment is that we do not know when to stop inserting each point to the buckets while guaranteeing every one of the top-k closest pairs appears at least in a bucket. Of course, if we insert each point to every bucket, we can get the gurantee, but we have to consider too many redundant pairs repeatedly for distance computation. Thus, we will introduce the safe condition of stopping early, called τ*-safety*, with the guarantee in the next section.

6.2.2 τ-Safety of Bucket Assignments

For a point p, let $d_{ij}^L(p)$ and $d_{ij}^R(p)$ be defined as follows:

$$d_{ij}^L(p) = \begin{cases} \infty, & \text{if } j = 1 \\ |s_{ij} - p(i)|, & \text{if } j \geq 2 \end{cases} \quad d_{ij}^R(p) = \begin{cases} \infty, & \text{if } j = n_i \\ |e_{ij} - p(i)|, & \text{if } j < n_i \end{cases}$$

where $p(i)$ denotes the i-th coordinate of p.

The $d_{ij}^L(p)$ and $d_{ij}^R(p)$ are the lower bound distances from the value of p in the i-dimension to the closest points outside of left and right boundaries of the range R_{ij} respectively. Since there is no point in the left side of R_{i1} and the right side of R_{in_i}, we let $d_{i1}^L(p) = \infty$ and $d_{in_i}^R(p) = \infty$.

For a given dimension i, consider a point p and a contiguous bucket set $B_i[j_i, k_i]$ = $\{B_{ij_i}, B_{i(j_i+1)}, \cdots, B_{ik_i}\}$ in which the range of a bucket includes p's i-th coordinate value. Then, we define $d_i^B(p) = \min(d_{ij_i}^L(p), d_{ik_i}^R(p))$. Note that $d_i^B(p)$ is the lower bound of the distance in the i-th dimension from the point p to the other points appearing at the outside of the bucket set in the left and right boundaries. If there is no bucket containing p in the i-th dimension, we simply let $d_i^B(p) = 0$.

For a point p, a set of contiguous bucket sets B_p=$\{B_1[j_1, k_1],\ B_2[j_2, k_2],\ \cdots,\ B_{\rho_p}[j_{\rho_p}, k_{\rho_p}]\}$, in which $B_i[j_i, k_i]$ represents a contiguous bucket set in the i-th dimension, is called a *safe bucket assignment* if for every dimension i s.t. $1 \leq i \leq \rho_p$, the range of a bucket in $B_i[j_i, k_i]$ always contains the p's i-th coordinate value and every bucket in $B_i[j_i, k_i]$ contains the point p. For a safe bucket assignment B_p for a point p, we let $d^B(p) = (\sum_{i=1}^{\rho_p} (d_i^B(p))^2)^{1/2}$ which represents the lower bound of the distance from p to the other points which do not appear in any bucket in B_p. For a given data D, a set of contiguous bucket sets B is called a *safe bucket assignment*, if B is a safe bucket assignment for every point $p \in D$. We next provide the following definition.

Definition 6.2.1 Given a set of points D with a space partitioning P, a safe bucket assignment for D with P is said to be τ-*safe* if every point in D satisfies the condition $d^B(p) \geq \tau$.

For the top-k closest pair problem with a given data set D, where the distances of the top-k closest pairs are at most τ, if we have a τ-*safe* bucket assignment, we will establish that we need to consider only the pairs of the points for distance computations within each bucket separately by the following lemma.

Lemma 6.2.2 *For the top-k closest pair problem with a given data set D, where the distances of the top-k closest pairs are at most τ, if we have a τ-*safe *bucket assignment using a given space partition, every pair of points whose distance is at most τ always appear together at least in a single bucket in the bucket assignment.*

Proof: We will prove the contrapositive of Lemma 6.2.2:

Suppose the safe bucket assignments of points p and q are included in the buckets of the first ρ_p dimensions and ρ_q dimensions respectively, where $1 \leq \rho_p \leq d$ and $1 \leq \rho_q \leq d$. Without loss of generality, we assume that $\rho_p \leq \rho_q$. If p and q do not share any bucket for every i-th dimension with $1 \leq i \leq \rho_p$, every range of the continuous bucket sets in every i-th dimension with $1 \leq i \leq \rho_p$ for p and q is not overlapped. Thus, the distance between the i-th coordinates of p and q (i.e., $|p(i) - q(i)|$) is at least $d_i^B(p)$. Since $|p(i) - q(i)| \geq d_i^B(p)$ holds with every i-th coordinate such that $1 \leq i \leq \rho_p$, the following inequality holds:

$$\begin{aligned} d(p,q) &= \left[\sum_{i=1}^{d} |p(i) - q(i)|^2\right]^{1/2} \geq \left[\sum_{i=1}^{\rho_p} |p(i) - q(i)|^2\right]^{1/2} \\ &\geq \left[\sum_{i=1}^{\rho_p} \left(d_i^B(p)\right)^2\right]^{1/2} = d^B(p) \geq \tau. \end{aligned} \tag{6.1}$$

In a τ-*safe* bucket assignment using a given space partition, if two points p and q in D do not appear together in any bucket, we showed the condition $d(p,q) \geq \tau$ with Equation (6.1). Thus, we have proved the contrapositive of Lemma 6.2.2. ∎

Example 6.2.3 Consider the point $p_1 = \langle 0.78, 0.4, 0.01 \rangle$ in Figure 6.3 and the space partitioning in Figure 6.7(a). Suppose that the order of dimensions is given as D_1, D_2, D_3 and $\tau = \sqrt{0.004}$. Since the value of p_1 in the first dimension is 0.78 and $0.78 \in R_{12}$ in Figure 6.7(a), the point p_1 should always appear in the bucket with R_{12} in the first dimension. After we insert p_1 into B_{12} as shown in Figure 6.7(b), since $d^B(p) = \sqrt{0.02^2} < \sqrt{0.004}$, we should select more buckets to satisfy τ-safety. For the next bucket, we can insert p into the one of the adjacent buckets B_{11} and B_{13}, or into the bucket B_{22} in the next dimension. If we select B_{13} as illustrated in Figure 6.7(c), we have $d^B(p_1) \geq \sqrt{0.004}$ and thus the safe bucket assignment $\{B_{12}, B_{13}\}$ satisfies τ-safety. The safe bucket assignment $\{B_{12}, B_{22}\}$ also satisfies τ-safety as illustrated in Figure 6.7(d).

Let us next consider the case of skipping any dimension without an insertion in

the given order of dimensions. Suppose we select the bucket set $\{B_{12}, B_{31}\}$ for p_1 satisfying $d^B(p_1) \geq \tau$, which includes the bucket in the third dimension without the bucket B_{22} in the second dimension. Suppose we have another point $p' = \langle 0.81, 0.41, 0.01 \rangle$ and we select the buckets $\{B_{13}, B_{22}\}$ satisfying $d^B(p') \geq \tau$. Then, they do not share any common bucket even though their distance $d(p_1, p') = \sqrt{0.001}$ is smaller than τ. ∎

Up to this point, we assumed that a space partitioning of each dimension and a specific order of dimensions are given. However, we will next discuss the issues on space partitioning and dimension ordering. Furthermore, we will also address how bucket selection needs to be done carefully.

6.2.3 Finding τ-safe Bucket Assignments

To find a τ-safe bucket assignment, we have to determine (1) how to split each dimension, (2) the order of dimensions for inserting a point and (3) how to insert each point in the buckets given the dimension order. Furthermore, we need (2) to compute an upper bound distance τ of the k-th closest pair in the given data in advance.

(1) How to split each dimension: We use the heuristic of splitting with the same number of ranges for every dimension such that every range contains almost same number of points. Using the method of Lagrange multiplier, we can prove that, for splitting n points into r partitions of which each partition has n_i points, $\sum_i^r n_i^2$ is minimized when $n_1 = n_2 = \cdots = n_r$ as shown in [87]. To split each dimension with even number of points, we compute the histogram with small size bins for each dimension and merge the bins so that each range has even number of points.

(2) The order of dimensions for inserting a point: Increasing $d^B(p)$ as much as we can in the earlier dimensions decreases the number of buckets for inserting a point p. Intuitively, if the points are distributed uniformly in a dimension i, the distance between the i-th coordinate of every point and the boundaries of the range of a safe contiguous range set in dimension i (i.e., $d_i^B(p)$) will become large. Thus, for deciding (2) the order of dimensions, we use the heuristic of sorting the dimensions with decreasing order of the variance of the values of the points in each dimension.

(3) How to insert each point in the buckets: Our heuristic is to insert a point into a bucket in every dimension one by one in the given order of dimensions until $d^B(p)$ becomes larger than τ. It is because inserting a point to a bucket in the next dimension is generally better in terms of computation than inserting it to a bucket with the neighboring ranges of previously inserted buckets. Suppose every point has to be included in 2 buckets to be τ-safe in a given space partitioning and each dimension is divided into r ranges with the same number of points for n points. If we insert the points in the neighboring buckets of dimension D_1 only, each bucket of D_1 will have about $2n/r$ points and it takes $O((2n/r)^2 \cdot r)$=$O(4n^2/r)$ time for distance computations. If we insert the points into a bucket in the first dimension and a bucket in the second dimension, each bucket has (n/r) points and it takes $O(2n^2/r)$ time. Thus, inserting every point into a bucket in every dimension one by one until $d^B(p)\geq\tau$ is better. However, when we finally reach the last dimension for insertion, $d^B(p)\geq\tau$ may not be still satisfied. In such a case, we repeatedly select a neighboring bucket, which increases $d^B(p)$ maximally, of a contiguous bucket set inserted previously in every dimension, and insert p into the selected bucket. We repeat this step until $d^B(p)\geq\tau$ is satisfied.

(4) Computation of an upper bound τ: To perform the essential pair partitioning, we need an upper bound distance τ of the k-th closest pair of the data points in advance. We will use a simple method of finding the k-th closest pair from sampled data points only with the size s by using a brute-force algorithm with $O(s^2)$ time and provide its distance as τ to our essential pair partitioning. Note that the distance of the k-th closest pair in any subset of the original data points is always larger or equal to that of the original data points. We found by experiments that the upper bound τ obtained from a small sample provides a reasonably good and tight upper bound.

6.2.4 Top-k Approximate Joins Using Essential Pair Partitioning

We now present the algorithms which find the top-k closest pairs utilizing essential pair partitioning combining with any top-k approximate join algorithm among *TopK-B* (brute-force), *TopK-D* (divide-and-conquer) and *TopK-T* (branch-and-bound) presented in Section 6.1. We first compute an upper bound τ of the k the closest pair by sampling the given vector data. We next generate a τ-safe bucket assignment for each point. Finally, we can compute the distances of pairs of points in each bucket due to Lemma 6.2.2 and return the top-k closest pairs among the top-k closest pairs from all buckets.

In Figure 6.8, we show the pseudocode of our top-k approximate join algorithm using both of essential pair partitioning and *TopK-T*. Given a set D of feature vectors generated from text documents, we first compute an upper bound τ of the k the closest pair by sampling the given vector data in lines 1–3. For each vector p in D, we next generates the τ-safe buckets of the vector p by invoking *GetSafeBucket* which returns the set of τ-safe buckets (lines 5–10). Then, we put the vector p into

Function TopK-FT.reduce(D, k)
D: the set of feature vectors
k: the number of pairs
begin
1. $sample$ = GetSample(D);
2. TOP_s = TopK-T($sample$, k);
3. $\tau = TOP_s$.getMax().dist;
4. B = the array of buckets;
5. **for each** p_i in D **do** {
6. $buckets$ = GetSafeBucket(p_i, τ);
7. **for each** a bucket index b in $buckets$ **do** {
8. Insert p_i into $B[b]$;
9. }
10. }
11. TOP_k = a max heap for ($dist$,$pair$) with $dist$ as keys
12. **for each** bucket index b in B **do** {
13. TOP_b = TopK-T($B[b]$, k);
14. **for each** $(dist, \langle p_i, p_j \rangle)$ in TOP_b **do** {
15. Insert $(dist, \langle p_i, p_j \rangle)$ into TOP_k;
16. **if** $|TOP_k| > k$ **then** DeleteMax(TOP_k);
17. }
18. }
19. **return** TOP_k;
end

Figure 6.8: The TopK-FT-MR algorithm

every bucket in the τ-safe buckets. After assigning with all vectors in D, we compute the top-k closest pairs in each bucket independently by utilizing *TopK-B* (lines 12–18). Note that instead of *TopK-B*, we can use either *TopK-B* or *TopK-D*. Finally, we aggregate the top-k closest pairs from every bucket and compute the top-k closest

Pid	τ-safe buckets
p_1	B_{12}, B_{22}
p_2	B_{11}
p_3	B_{12}
p_4	B_{11}
p_5	B_{12}, B_{22}, B_{31}
p_6	B_{13}, B_{22}

Figure 6.9: The τ-bucket assignments when τ=0.09

pairs.

Example 6.2.4 Consider the 3-dimensional points in Figure 6.3. Assume that the threshold τ is 0.09, the order of dimension is D_1, D_2, D_3 and the space partitioning with 3 dimensions is done as shown in Figure 6.7(a). Let us first compute the τ-safe buckets of p_5=$\langle 0.77, 0.42, 0.02\rangle$ in Figure 6.3. The buckets containing p_5 in all 3 dimensions are B_{12}, B_{22} and B_{31}. With only B_{12} in the first dimension, $d^B(p_5)$ is $(\min(|0.77 - 0.4|^2, |0.77 - 0.8|^2))^{1/2}$=0.03, which is smaller than τ. Considering the bucket B_{22} in the next dimension together, $d^B(p_5)$ is $(0.0009 + \min(|0.42 - 0.3|^2, |0.42 - 0.5|^2))^{1/2} = \sqrt{0.0073}$ which is still smaller than τ. With B_{31} in the third dimension, $d^B(p_5)$ is $\sqrt{0.0137}$, which is larger than τ, and finally, the τ-safe buckets of p_5 is $\{B_{12}, B_{22}, B_{31}\}$.

Let us next consider the point p_2=$\langle 0.07, 0.21, 0.57\rangle$ in Figure 6.3. The bucket of the first is B_{11} with the partitioning in Figure 6.7(a). With B_{11} only, $d^B(p_2)$ is $\min(\infty, |0.07 - 0.3|^2)$ = 0.21, which is already larger than τ = 0.09. Thus, the τ-safe bucket of p_2 is $\{B_{11}\}$.

Similarly, if we compute the τ-safe bucket assignments for the rest of the points in Figure 6.3, we get the τ-safe bucket assignments as shown in Figure 6.9(a). If we compute the distances of all pairs of points in each bucket, we perform only 7 distance computations with *essential pair partitioning* while the brute-force algorithm *TopK-PB- MR* need 15 distance computations. Among the top-2 closest pairs from each bucket, the overall top-2 closest pairs are (p_1, p_5) and (p_1, p_6). ∎

6.3 Experiments

We empirically compared the performances of our proposed algorithms. All experiments were performed on the Intel(R) Core(TM)2 Duo CPU 2.66GHz machine with 2GB of main memory running Linux operating systems. All algorithms were implemented using Javac Compiler of version 1.6. In all of our experiments, we use Euclidean distance as the distance measure. The execution times in the graphs shown in this section are plotted in a log scale.

6.3.1 Implemented Algorithms

We implemented the following algorithms:

- **TopK-B:** It is the brute-force algorithm which finds the top-k closest pairs with $O(d \cdot n^2)$ time.
- **TopK-D:** This is the divided-and-conquer algorithm *TopK-D* proposed in Section 6.1.
- **TopK-T:** It is the implementation of the branch-and-bound algorithm *TopK-B* proposed in Section 6.1.
- **TopK-FB:** We implemented the algorithm using both of essential pair partitioning presented in Section 6.2 and the brute-force algorithm *TopK-B*.
- **TopK-FD:** This is the implementation of the algorithm using both essential pair partitioning and the algorithm *TopK-D*.
- **TopK-FT:** It represents the algorithm utilizing both essential pair partitioning and our algorithm *TopK-T*.

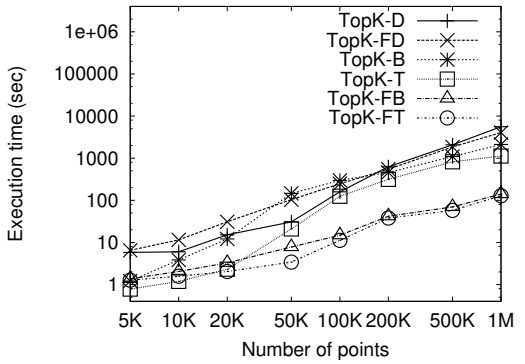
6.3.2 Data Sets

Synthetic data: We generated synthetic data with the points of which each dimension has a real number between 0 and 1. To produce realistic synthetic data sets, we enforce our synthetic data sets to follow the Zipfian distribution as in [67], in which the sizes of clusters containing the pairs of points within distances less than 0.01 follow the Zipfian distribution $\eta(x) = \frac{10^3}{(x+1)^\gamma}$ where x is the rank on the size and γ is a parameter determining the distribution of the points. We generate the synthetic data with $\gamma = 1.5$ as the default value. Furthermore, we generate several synthetic data sets with varying the number of points from 5,000 to 1,000,000. We also varied the number of dimensions from 2 to 100 with the data sets consisting of 10,000 points.

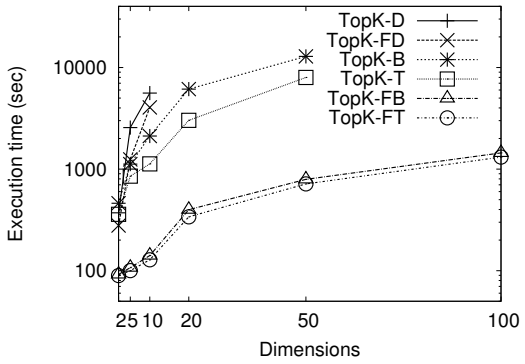
Real-life data: We downloaded the *COREL* image feature data from the UCI Machine Learning Repository[2]. This data contains the image features extracted from a Corel image collection. There are 68,040 points, which represent the color histogram of photo images and every point has 32 dimensions of real numbers.

6.3.3 Performance Results with Synthetic Data

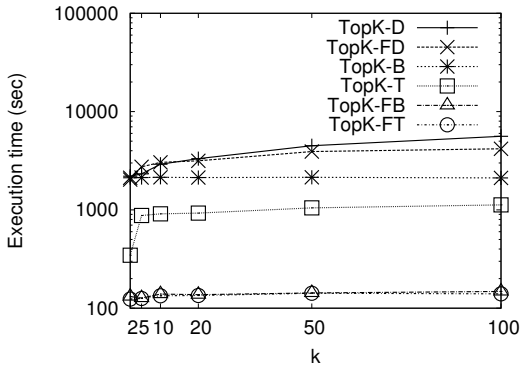
We now present our experimental results with synthetic data. In the experiments, we varied three parameters: the number of closest pairs k and the number of dimensions d and the number of vectors n. The default parameters are: the number of data points n=50,000, the number of dimensions d=10 and the number of pairs to find k=50. For essential pair partitioning, we use default value of r=60, which is the number of ranges in every dimension.



(a) Varying n



(b) Varying d



(c) Varying k

Figure 6.10: Execution times with varying n, d and k

Varying n: We varied n from 5,000 to 1,000,000 and the execution times are plotted in Figure 6.10(a). Note that both of the x-axis and y-axis are in a log scale. For the range of data sizes from 5,000 to 10,000, our algorithm *TopK-T* without using essential pair partitioning shows the best performance. However, when data sizes are larger than 20,000, *TopK-FT* is the best performer and is about 2 times faster than *TopK-T*. Furthermore, when data sizes are larger than 50,000, *TopK-FB* becomes better than *TopK-T*. Thus, we can see that our essential pair partitioning is very effective for finding top-k closest pairs with large data.

Varying d: In the following experiments, we varied the number of dimension d from 2 to 100. The execution times are plotted in Figure 6.10(b). We did not plot the execution times of *TopK-D* and *TopK-FD* with $d > 10$ because they do not stop within 2 hours. For the data set with d between 2 to 10, *TopK-T* is the fastest. However, The graph show that *TopK-FB* is the fastest algorithm. With increasing d, the relative speed of *TopK-FB* to the other algorithms without essential pair partitioning becomes faster gradually and when d=50, it becomes 11.2 times faster than *TopK-T*. This result illustrates that *TopK-T* is the best for small dimensions and the essential pair partitioning is very effective for higher dimensions.

Varying k: With varying k from 2 to 100, we plotted the execution times in Figure 6.10(c). With a wide range of k, *TopK-FT* outperforms the other algorithms. With all ranges of k, *TopK-FT* is about 40 times faster than *TopK-B* and 5 times faster than *TopK-D*. Thus, we can see that essential pair partitioning is very effective, and *TopK-FT* and *TopK-FB* are not influenced much by k.

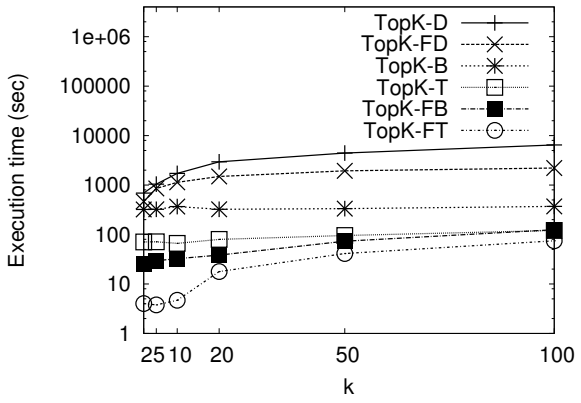


Figure 6.11: Performance result with COREL

6.3.4 Performance Results with Real-life Data

We present our experimental results with the real-life data COREL with $68,040$ number of points with 32 dimensions. With varying k from 2 to 100, we plotted the execution times in Figure 6.11. The y-axis is in a log scale. With every k, our algorithm *TopK-FT* shows the best performance. *TopK-FT* is faster than *TopK-FB*, which is the second best performer, at least by 1.78 times. *TopK-D* shows the worst performance due to its time complexity which is exponential to the dimensionality.

6.3.5 Similarity Query Processing by Integrating Our All Proposed Techniques

As we discussed in Chapter 1, realistic and practical applications frequently contain similarity joins as well as selection conditions represented by substring matching. Actually, many commercial search engines such as Google[29], Bing[11] and Yahoo![95] provide query results obtained by utilizing all those similarity queries together. Thus, we next evaluate the performance of processing such similarity queries

by integrating the algorithms proposed in this dissertation.

Recall that given a query string, our motivating application of a web-based search engine introduced in Chapter 1 provides a query result by grouping the relevant documents, which are retrieved by exact and top-k approximate substring matching for the query string, according to their similarities. In our experiment, we implemented query processing algorithms which perform similarity joins for the *DBLP* titles obtained by selection conditions which are represented exact or approximate substring matching.

Real-life data: We utilize the *DBLP titles* consisting of 1,000,000 titles collected from DBLP [51]. We also transform each DBLP title to a 100-dimensional vector, where the range of every coordinate is between 0 and 1, by performing a dimension reduction using PLSI[32] with 100 topics.

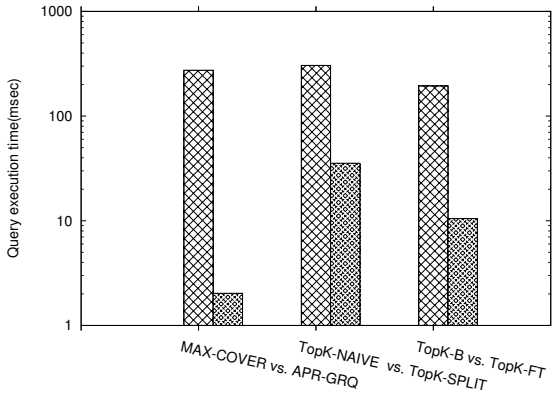
Test query: We randomly sampled 50 query strings from the name entity data collected for Name Entity Recognition in [20] and the lengths of the query strings are between 5 to 26.

Implemented SQL query: Let *dblp* be a table with a scheme *dblp(id,title)* where *id* and *title* are the ID and title respectively. Let $d_{sub}(s_1, s_2)$ denote the substring edit distance between two strings s_1 and s_2 introduced in Section 3.2. Furthermore, for a title with id, $fvector(id)$ denotes its feature vector which is generated by PLSI[32] in advance to the actual query processing. Finally, $d_2(v_1, v_2)$ represents the Euclidean distance between two vectors v_1 and v_2. For a given query string σ, the SQL query to be tested in our experiments is following.

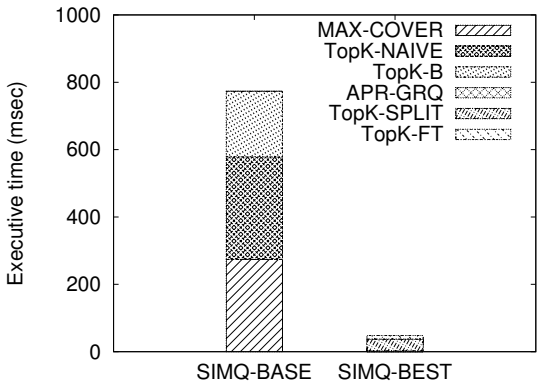
```
1: SELECT *
2:  FROM
3:     (SELECT * FROM dblp
4:      WHERE title LIKE '%σ%' LIMIT 1000)
5:  UNION
6:     (SELECT * FROM dblp
7:      ORDER BY d_sub(title,σ) ASC LIMIT 1000)
8:  ORDER BY d_2(fvector(id),fvector(id)) ASC
9:  LIMIT 10
```

Implemented algorithms: To process the above SQL statement, we assume that the optimal query plan is to first perform the two nested selection queries one by one and then next execute the top-k similarity join with the query results of the selection queries. We implemented the following algorithms.

- **SIMQ-BASE:** This is an implementation by integrating the following naive (or traditional) algorithms: For exact substring matching in lines 3–4 of the above SQL statement, we used the naive algorithm *MAX-COVER* in Chapter 4. For top-k approximate substring matching in lines 6–7, we utilized the naive algorithm *TopK-NAIVE* in Chapter 5. For top-k similarity join, the brute-force algorithm *TopK-B* in Chapter 6 was used.
- **SIMQ-BEST:** We implemented a query processing algorithm using our best algorithms as follows: For exact substring matching in lines 3–4 of the SQL statement, we used the best performer *APR-GRQ* in Chapter 4. For top-k approximate substring matching in lines 6–7, we utilized the best algorithm *TopK-SPLIT* in Chapter 5. For top-k similarity join, the most efficient algorithm *TopK-FT* in Chapter 6 was used.



(a) Execution time of each step



(b) Execution times of query processing

Figure 6.12: Performance result of our integrated query processing technique

Execution times: We first plotted the execution times for each step of *SIMQ-BASE* and *SIMQ-BEST* separately in Figure 6.12(a). The graph shows that all our proposed algorithms outperform the naive (or traditional) algorithms. We next plotted total execution times of *SIMQ-BASE* and *SIMQ-BEST* in Figure 6.12(b). The graph confirms that our proposed query processing algorithm *SIMQ-BEST* is faster than the baseline algorithm *SIMQ-BASE* by 9.58 times.

Chapter 7

Conclusion

Similarity query processing on text data plays seminal roles in many applications such as web search engines. In this dissertation, we investigated the efficient processing of three important similarity queries which are exact substring matching, top-k approximate substring matching and top-k approximate string joins.

First, we studied the problem of exact substring matching. We proposed the optimal algorithms to find optimal plans by taking advantage of the inverted indexes using variable-length grams. We developed the optimal algorithms *OPT-QSP* as well as the efficient approximate algorithm *APR-GRQ* to overcome the exponential nature of search space for finding the optimal query plan. In our experiments, we showed that our optimal and approximate algorithms improve query execution time significantly.

Second, we investigated the problem of top-k approximate substring matching. We developed novel filtering methods using q-grams which enable us to filter out many strings without computing the actual substring edit distances to the query string. Then, we presented two algorithms *TopK-LB* and *TopK-SPLIT* which efficiently

find top-k approximate substring matches by utilizing our proposed filtering techniques. In our experiments, we showed that our *TopK-FT* is very efficient and scale well with large text data.

Finally, we examined the problem of top-k approximate string joins. We developed the algorithm *TopK-T* which computes top-k similarity joins efficiently. Furthermore, we proposed the *essential pair partitioning* technique, which enables us to ignore many of document pairs which cannot not participate in the top-k join result. Then, we devised the improved algorithm *TopK-FT* which is a generalization of *TopK-T* based on the essential pair partitioning method. In our experiments, we showed that our proposed algorithm and essential pair partitioning are very efficient.

The similarity queries studied in this dissertation take very important rolls in many applications such as web search engines and our proposed algorithms can be practically used in such applications to improve the performance. We next discuss the direction of future work.

Parallelization of the similarity query processing: The text query processing algorithms proposed in this dissertation show nice scalability for large text data. However, due to the explosive growth in the amount of data created in the Internet, text query processing becomes even more challenging today. Recently, for such data-intensive computations, the MapReduce[24] paradigm has recently received a lot of attention. MapReduce is a programming model for easy development of scalable parallel applications to process vast amounts of data on large clusters of commodity machines. Due to its characteristics of using shared-nothing architecture, it is important to minimize the amount of data to shared between the clusters of machines to reduce network traffic. Thus, the partitioning based lower bound proposed in

Chapter 5 can be used in the MapReduce framework usefully since it can compute the lower bound of substring edit distance for each string independently. The essential pair partitioning presented in Chapter 6 is also useful to use in MapReduce and thus we developed the parallel top-k approximate join algorithms using MapReduce in [41].

Processing genetic sequence matching queries: Similarity queries for biological genetic sequences have been also extensively studied[17, 18, 45, 59]. Since DNA sequences come from a smaller alphabet than text data in natural languages, we need to develop efficient string matching algorithms based on the assumption that the size of alphabet is small. Furthermore, for approximate substring matching in genetic sequences, they use different similarity measurements such as k-mismatch distance[56], BWT distance[57] and rank distance[25]. Thus, we need to devise new algorithms which can process the genetic sequence matching queries with those similarity measurements efficiently.

For each type of similarity queries, we performed extensive experiments, and confirmed the effectiveness and scalability of our query processing algorithms. Furthermore, we also showed that the practical query used for our motivating application in Chapter 1 is efficiently processed with our proposed algorithms. We believe that our algorithms proposed in this dissertation can be applied practically in many important applications and enhance the performance of similarity query processing.

References

[1] A. Andoni and K. Onak. Approximating edit distance in near-linear time. In *STOC*, 2009.

[2] U. K. Archive. Uci kdd archive. http://kdd.ics.uci.edu/, 2010.

[3] R. A. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[4] R. A. Baeza-Yates and B. A. Ribeiro-Neto. *Modern Information Retrieval - the concepts and technology behind search, Second edition*. Pearson Education Ltd., Harlow, England, 2011.

[5] A. Bagga and B. Baldwin. Entity-based cross-document coreferencing using the vector space model. In *COLING-ACL*, 1998.

[6] R. J. Bayardo, Y. Ma, and R. Srikant. Scaling up all pairs similarity search. In *International World Wide Web Conference*, pages 131–140, 2007.

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

[8] A. Behm, S. Ji, C. Li, and J. Lu. Space-constrained gram-based indexing for efficient approximate string search. In *ICDE*, 2009.

[9] J. L. Bentley and M. I. Shamos. Divide-and-conquer in multidimensional space. In *8th Annual ACM Symposium on the Theory of Computing*, pages 220–230, 1976.

[10] M. Bilenko, R. J. Mooney, W. W. Cohen, P. Ravikumar, and S. E. Fienberg. Adaptive name matching in information integration. *IEEE Intelligent System*, 18(5):16–23, 2003.

[11] Bing. http://www.bing.com/.

[12] D. M. Blei, A. Y. Ng, and M. I. Jordan. Latent dirichlet allocation. *Journal of Machine Learning Research*, 3:993–1022, 2003.

[13] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. *Computer Networks*, 30(1-7):107–117, 1998.

[14] A. Z. Broder. On the resemblance and containment of documents. In *Proceedings of Compression and Complexity of SEQUENCES*, pages 21–29, 1997.

[15] S. Chaudhuri, V. Ganti, and L. Gravano. Selectivity Estimation for String Predicates: Overcoming the Underestimation Problem. In *IEEE International Conference on Data Engineering*, pages 227–238, 2004.

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

[17] F. Claude, A. Fariña, M. A. Martínez-Prieto, and G. Navarro. Compressed q-gram indexing for highly repetitive biological sequences. In *IEEE International Conference on Bioinformatics and Bioengineering*, pages 86–91, 2010.

[18] F. Claude, G. Navarro, H. Peltola, L. Salmela, and J. Tarhio. String matching with alphabet sampling. *Journal of Discrete Algorithms*, 11:37–50, 2012.

[19] E. Cohen. Size-estimation framework with applications to transitive closure and reachability. *J. Comput. Syst. Sci.*, 55(3):441–453, 1997.

[20] CoNLL-2003. http://www.cnts.ua.ac.be/conll2003/ner/, 2003.

[21] A. Corral, Y. Manolopoulos, Y. Theodoridis, and M. Vassilakopoulos. Closest pair queries in spatial databases. In *SIGMOD*, pages 189–200, 2000.

[22] A. Corral, Y. Manolopoulos, Y. Theodoridis, and M. Vassilakopoulos. Algorithms for processing k-closest-pairs queries in spatial databases. *Data & Knowledge Engineering*, 49(1):67–104, 2004.

[23] T. Cox and M. Cox. *Multidimensional Scaling. Monographs on Statistics and Applied Probability*. Chapman & Hall/CRC, Boca Raton, 2001.

[24] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI*, 2004.

[25] L. P. Dinu and A. Sgarro. A low-complexity distance for dna strings. *Fundam. Inform.*, 73(3):361–372, 2006.

[26] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. *Journal of Computer and System Sciences*, 66(4):614–656, 2003. Special Issue on PODS 2001.

[27] C. Faloutsos, R. T. Ng, and T. K. Sellis. Flexible and adaptable buffer management techniques for database management systems. *IEEE Trans. Computers*, 44(4):546–560, 1995.

[28] P. Ganesan, H. Garcia-Molina, and J. Widom. Exploiting hierarchical domain structure to compute similarity. *ACM Trans. Inf. Syst.*, 21(1):64–93, 2003.

[29] Google. http://www.google.com/.

[30] J. M. Hellerstein, E. Koutsoupias, and C. H. Papadimitriou. On the analysis of indexing schemes. In *PODS*, pages 249–256, 1997.

[31] D. T. Hoang, P. M. Long, and J. S. Vitter. Dictionary selection using partial matching. *Information Sciences*, 119(1-2):57–72, 1999.

[32] T. Hofmann. Probabilistic latent semantic indexing. In *SIGIR*, 1999.

[33] T. Hofmann. Probabilistic topic maps: Navigating through large text collections. In *IDA*, pages 161–172, 1999.

[34] T. Hofmann and J. Puzicha. Statistical models for co-occurrence data. In *Massachusetts Institute of Technology, Technical report*, 1998.

[35] H. V. Jagadish, R. T. Ng, and D. Srivastava. Substring selectivity estimation. In *Proceedings of ACM Symposium on Principles of Database Systems*, pages 249–260, 1999.

[36] L. Jin and C. Li. Selectivity estimation for fuzzy string predicates in large data sets. In *Proceedings of the Conference on Very Large Data Bases*, pages 397–408, 2005.

[37] L. Jin and C. Li. Selectivity estimation for fuzzy string predicates in large data sets. In *VLDB*, 2005.

[38] I. Jolliffe. *Principal Component Analysis, second edition*. Springer, New York, 2002.

[39] N. Katoh and K. Iwano. Finding k farthest pairs and k closest/farthest bichromatic pairs for points in the plane. In *8th Annual Computational Geometry*, 1992.

[40] M. Kim, K. Whang, J. Lee, and M. Lee. n-Gram/2L: A space and time efficient two-level n-gram inverted index structure. In *Proceedings of the Conference on Very Large Data Bases*, pages 325–336, 2005.

[41] Y. Kim and K. Shim. Parallel top-k similarity join algorithms using mapreduce. In *ICDE*, pages 510–521, 2012.

[42] Y. Kim, K.-G. Woo, H. Park, and K. Shim. Efficient processing of substring match queries with inverted q-gram indexes. In *IEEE International Conference on Data Engineering*, pages 721–732, 2010.

[43] D. E. Knuth. *The Art of Computer Programming, Volume III: Sorting and Searching*. Addison-Wesley, 1973.

[44] N. Koudas, A. Marathe, and D. Srivastava. Flexible string matching against large databases in practice. In *VLDB*, 2004.

[45] K. Laurio, F. Linåker, and A. Narayanan. Regular biosequence pattern matching with cellular automata. *Information Sciences*, 146(1-4):89–101, 2002.

[46] D. D. Lee and H. S. Seung. Algorithms for non-negative matrix factorization. In *NIPS*, 2000.

[47] H. Lee, R. T. Ng, and K. Shim. Extending q-grams to estimate selectivity of string matching with low edit distance. In *Proceedings of the Conference on Very Large Data Bases*, pages 195–206, 2007.

[48] H. Lee, R. T. Ng, and K. Shim. Approximate substring selectivity estimation. In *EDBT*, 2009.

[49] H. P. Lenhof and M. Smid. Sequential and parallel algorithms for the k closest pairs problem. *International Journal of Computational Geometry and Applications*, pages 273–288, 1995.

[50] V. I. Levenshtein. Binary codes capable of correcting deletions, insertions and reversals. *Soviet Phys. Dokl.*, 1985.

[51] M. Ley. Dblp computer science bibliography. http://dblp.uni-trier.de/xml/, 2011.

[52] C. Li, J. Lu, and Y. Lu. Efficient merging and filtering algorithms for approximate string searches. In *ICDE*, 2008.

[53] C. Li, B. Wang, and X. Yang. Vgram: Improving performance of approximate queries on string collections using variable-length grams. In *Proceedings of the Conference on Very Large Data Bases*, pages 303–314, 2007.

[54] G. Li, D. Deng, and J. Feng. Faerie: efficient filtering algorithms for approximate dictionary-based entity extraction. In *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, pages 529–540, 2011.

[55] C.-C. Liu, J.-L. Hsu, and A. L. P. Chen. An approximate string matching algorithm for content-based music data retrieval. In *ICMCS*, 1999.

[56] Z. Liu, X. Chen, J. Borneman, and T. Jiang. A fast algorithm for approximate string matching on gene sequences. In *CPM*, pages 79–90, 2005.

[57] S. Mantaci, A. Restivo, and M. Sciortino. Distance measures for biological sequences: Some recent approaches. *Int. J. Approx. Reasoning*, 47(1):109–124, 2008.

[58] A. Mazeika, M. H. Böhlen, N. Koudas, and D. Srivastava. Estimating the selectivity of approximate string queries. *ACM Trans. Database Syst.*, 32(2), 2007.

[59] C. Miao, G. Chang, and X. Wang. Filtering based multiple string matching algorithm combining q-grams and bndm. In *International Conference on Genetic and Evolutionary Computing (ICGEC)*, pages 582–585, 2010.

[60] L. Monier. Combinatorial solutions of multidimensional divide-and-conquer recurrences. *Journal of Algorithms*, 1(1):60–74, 1980.

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

[62] G. Navarro and R. A. Baeza-Yates. A practical q -gram index for text retrieval allowing errors. *CLEI Electron. J.*, 1(2), 1998.

[63] 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.

[64] G. Navarro, E. Sutinen, and J. Tarhio. Indexing text with approximate -grams. *J. Discrete Algorithms*, 3(2-4):157–175, 2005.

[65] Y. Ogawa and T. Matsuda. Optimizing query evaluation in n-gram indexing. In *Proceedings of the international ACM SIGIR conference on Research and development in information retrieval*, pages 367–368, 1998.

[66] R. Ostrovsky and Y. Rabani. Low distortion embeddings for edit distance. *J. ACM*, 54(5), 2007.

[67] C. R. Palmer and C. Faloutsos. Density biased sampling: an improved method for data mining and clustering. *SIGMOD Rec.*, 29(2):82–92, 2000.

[68] C. Platzer and S. Dustdar. A vector space search engine forweb services. In *ECOWS*, pages 62–71, 2005.

[69] M. Sahami and T. D. Heilman. A web-based kernel function for measuring the similarity of short text snippets. In *WWW*, 2006.

[70] J. S. Salowe. Enumerating interdistances in space. *International Journal of Computational Geometry and Applications*, 1991.

[71] G. Salton. Full text information processing using the smart system. *IEEE Data Eng. Bull.*, 13(1):2–9, 1990.

[72] S. Santini and R. Jain. Similarity measures. *IEEE Trans. Pattern Anal. Mach. Intell.*, 1999.

[73] S. Sarawagi and A. Kirpal. Efficient set joins on similarity predicates. In *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, pages 743–754, 2004.

[74] S. Sarawagi and A. Kirpal. Efficient set joins on similarity predicates. In *SIGMOD*, 2004.

[75] F. Scholer, H. E. Williams, J. Yiannis, and J. Zobel. Compression of inverted indexes for fast query evaluation. In *Proceedings of the international ACM SIGIR conference on Research and development in information retrieval*, pages 222–229, 2002.

[76] P. H. Sellers. The theory and computation of evolutionary distances: Pattern recognition. *J. Algorithms*, 1(4), 1980.

[77] A. Silberschatz, H. F. Korth, and S. Sudarshan. *Database System Concepts*. McGraw-Hill, Inc., New York, NY, USA, 2002.

[78] S. Suri. Closest pair problem. http://www.cs.ucsb.edu/šuri/cs235/ClosestPair.pdf, 2002.

[79] Times. Times online. http://www.timesonline.co.uk/tol/news/.

[80] B. S. K. Tsuda and J. Vert. *Kernel Methods in Computational Biology*. MIT Press, Cambridge, 2004.

[81] H. R. Turtle and J. Flood. Query evaluation: Strategies and optimizations. *Inf. Process. Manage.*, 31(6):831–850, 1995.

[82] Twitter. http://twitter.com.

[83] E. Ukkonen. Finding approximate patterns in strings. *J. Algorithms*, 6(1):132–137, 1985.

[84] R. Vernica and C. Li. Efficient top-k algorithms for fuzzy search in string collections. In *KEYS*, 2009.

[85] W. Wang, C. Xiao, X. Lin, and C. Zhang. Efficient approximate entity extraction with edit distance constraints. In *SIGMOD*, 2009.

[86] Wikipedia. http://en.wikipedia.org.

[87] Wikipedia. Lagrange multipliers. http://en.wikipedia.org/wiki/ Lagrange_multipliers, 2010.

[88] Wikipedia. Binomial distribution. http://en.wikipedia.org/wiki/ Binomial_distribution, 2011.

[89] Wikipedia. Database download. http://en.wikipedia.org/wiki/ Wikipedia:Database_download, 2011.

[90] Wikipedia. Negative binomial distribution. http://en.wikipedia.org/wiki/ Negative_binomial_distribution, 2011.

[91] C. Xiao, W. Wang, and X. Lin. Ed-join: an efficient algorithm for similarity joins with edit distance constraints. *PVLDB*, 2008.

[92] C. Xiao, W. Wang, X. Lin, and H. Shang. Top-k set similarity joins. In *ICDE*, 2009.

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

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

[95] Yahoo. http://www.yahoo.com/.

[96] X. Yang, B. Wang, and C. Li. Cost-based variable-length-gram selection for string collections to support approximate queries efficiently. In *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, pages 353–364, 2008.

[97] Z. Yang, J. Yu, and M. Kitsuregawa. Fast algorithms for top-k approximate string matching. In *Association for the Advancement of Artificial Intelligence*, pages 1467–1473, 2010.

[98] P. Zezula, G. Amato, V. Dohnal, and M. Batko. *Similarity Search - The Metric Space Approach*, volume 32 of *Advances in Database Systems*. Kluwer, 2006.

[99] Z. Zhang, M. Hadjieleftheriou, B. C. Ooi, and D. Srivastava. Bed-tree: an all-purpose index structure for string similarity search based on edit distance. In *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, pages 915–926, 2010.

[100] Y. Zhao and G. Karypis. Empirical and theoretical comparisons of selected criterion functions for document clustering. *Machine Learning*, 55(3):311–331, 2004.

[101] N. K. Zhiyuan Chen, Flip Korn and S. Muithukrishnan. Selectivity Estimation For Boolean Queries. In *Proceedings of ACM Symposium on Principles of Database Systems*, pages 216–225, 2000.

[102] M. Zhu, D. Papadias, J. Zhang, and D. L. Lee. Top-k spatial joins. *TKDE*, 17(4), 2005.

[103] J. Zobel and P. W. Dart. Finding approximate matches in large lexicons. *Softw., Pract. Exper.*, 25(3):331–345, 1995.

[104] J. Zobel and A. Moffat. Inverted files for text search engines. *ACM Computing Surveys*, 38(2):–, 2006.

[105] J. Zobel, A. Moffat, and R. Sacks-davis. Searching large lexicons for partially specified terms using compressed inverted files. In *Proceedings of the Conference on Very Large Data Bases*, pages 290–301, 1993.

요 약 문

텍스트 데이터에 대한 유사도질의 처리기술

인터넷이 널리 사용되기 시작하며 수많은 텍스트 데이터가 생성되고 텍스트 안에서 효율적으로 유사 문자열을 찾는 방법 또한 중요한 문제가 되었다. 여러 응용 프로그램에서 유사도 질의는 필수적이며 매우 유용한 부분을 차지하고 있다. 기존의 유사도 질의는 사용자가 정한 문턱값에 대해서 질의 문자열과 문서의 유사도가 주어진 문턱값보다 작은 모든 문서를 찾는 접근 방법을 사용하였다. 그러나 사용자가 적당한 문턱값을 미리 알 수가 없기 때문에 적당한 값을 찾기 위해 여러 번 다른 값을 이용해 질의를 해봐야 하는 문제가 있다. 따라서 문턱값을 사용하지 않고 유사한 K개의 문서를 찾는 방법이 최근 활발히 연구되고 있다.

본 논문에서 우선 Google, Bing, Yahoo!과 같은 웹 검색엔진을 비롯한 많은 응용 프로그램에서 실질적으로 유용하게 사용되는 유사도 질의에 대한 대표적인 응용 프로그램을 제시한다. 이 유사도 질의를 효율적으로 처리하기 위해서는 부분 문자열 검색, K근사 부분 문자열 검색 그리고 K근사 문자열 조인에 대한 빠른 질의 처리 방법이 필요하다. 따라서 우리는 이러한 유사도 질의들을 효율적으로 처리하는 문제에 대해 연구하고자 한다.

부분 문자열 검색을 위해 가변 길의 Q그램을 사용한 역 색인을 이용하여 최적의 질의 계획을 찾는 최적(optimal) 알고리즘을 우선 소개하였다. 또한 최적의 질의

계획을 찾기 위해 고려해야 하는 경우의 수가 기하급수적으로 크기 때문에 최적에 근사한 질의 계획을 찾는 근사(approximate) 알고리즘도 개발하였다. K근사 부분 문자열 검색을 위해서는 본 연구에서 제안한 부분문자열 최단편집길이(edit distance)의 최소한도(lower bound) 계산법을 이용한 필터링 기술을 개발하였다. 또한 이 필터링 기술을 응용한 효율적인 K근사 문자열 검색 알고리즘을 제안하였다. 다음으로 우리는 K근사 문자열 조인을 위해 조인에 고려해야 하는 다수의 문서 쌍을 미리 제거할 수 있는 분할(partitioning) 기술을 제안하고 이를 이용한 효율적인 알고리즘을 제안하였다. 그리고 실험을 통해 본 논문에서 제안한 알고리즘들이 매우 효과적으로 질의 처리 비용을 줄일 수 있음을 확인하였다.

본 논문에서 연구한 유사도 질의는 웹 검색엔진과 같은 많은 응용 프로그램에 필요하므로 우리가 제안한 알고리즘이 그러한 응용 프로그램의 성능을 효과적으로 향상시킬 수 있을 것이라 생각한다.

주요어: *유사도 질의, 부분 문자열 검색, K근사 부분 문자열 검색, K근사 문자열 조인*
학번: 2007-20948