

FINDING SETS OF HIGH-FREQUENCY QUERIES FOR HIGH-FREQUENCY-QUERY-
BASED FILTER FOR SIMILARITY JOIN

Miss Kamolwan Kunanusont



จุฬาลงกรณ์มหาวิทยาลัย

CHULALONGKORN UNIVERSITY

บทคัดย่อและแฟ้มข้อมูลฉบับเต็มของวิทยานิพนธ์ตั้งแต่ปีการศึกษา 2554 ที่ให้บริการในคลังปัญญาจุฬาฯ (CUIR)
เป็นแฟ้มข้อมูลของนิสิตเจ้าของวิทยานิพนธ์ ที่ส่งผ่านทางบัณฑิตวิทยาลัย

The abstract and full text of theses from the academic year 2011 in Chulalongkorn University Intellectual Repository (CUIR)
are the thesis authors' files submitted through the University Graduate School.

A Thesis Submitted in Partial Fulfillment of the Requirements
for the Degree of Master of Science Program in Computer Science and Information
Technology

Department of Mathematics and Computer Science

Faculty of Science

Chulalongkorn University

Academic Year 2014

Copyright of Chulalongkorn University

การค้นหาเซตของข้อความที่ใช้อยู่สำหรับตัวกรองข้อความที่ใช้อยู่สำหรับการเชื่อมด้วยสาย
อักขระคล้าย



วิทยานิพนธ์นี้เป็นส่วนหนึ่งของการศึกษาตามหลักสูตรปริญญาวิทยาศาสตรมหาบัณฑิต
สาขาวิชาวิทยาการคอมพิวเตอร์และเทคโนโลยีสารสนเทศ ภาควิชาคณิตศาสตร์และวิทยาการ
คอมพิวเตอร์

คณะวิทยาศาสตร์ จุฬาลงกรณ์มหาวิทยาลัย

ปีการศึกษา 2557

ลิขสิทธิ์ของจุฬาลงกรณ์มหาวิทยาลัย

Thesis Title	FINDING SETS OF HIGH-FREQUENCY QUERIES FOR HIGH-FREQUENCY-QUERIES-BASED FILTER FOR SIMILARITY JOIN
By	Miss Kamolwan Kunanusont
Field of Study	Computer Science and Information Technology
Thesis Advisor	Assistant Professor Dr. Jaruloj Chongstitvatana

Accepted by the Faculty of Science, Chulalongkorn University in Partial
Fulfillment of the Requirements for the Master's Degree

.....Dean of the Faculty of Science
(Professor Supot Hannongbua, Dr.rer.nat)

THESIS COMMITTEE

.....Chairman
(Assistant Professor Dr. Saranya Maneeroj)

.....Thesis Advisor
(Assistant Professor Dr. Jaruloj Chongstitvatana)

.....External Examiner
(Doctor Kamol Keatruangkamala)

กมลวรรณ คุณานุสนธิ : การค้นหาเซตของข้อความที่ใช้บ่อยสำหรับตัวกรองอิงข้อความที่ใช้บ่อยสำหรับการเชื่อมด้วยสายอักขระคล้าย (FINDING SETS OF HIGH-FREQUENCY QUERIES FOR HIGH-FREQUENCY-QUERY-BASED FILTER FOR SIMILARITY JOIN)
 อ.ที่ปรึกษาวิทยานิพนธ์หลัก: ผศ. ดร. จารุโลจน์ จงสถิตย์วัฒนา, 75 หน้า.

การค้นหาด้วยสายอักขระคล้าย และการเชื่อมด้วยสายอักขระคล้ายเป็นตัวดำเนินการที่สำคัญในฐานข้อมูลสายอักขระ การค้นหาด้วยสายอักขระคล้ายจะคืนค่าระเบียบในตารางที่คล้ายกับข้อความ คำถาม ในขณะที่การเชื่อมด้วยสายอักขระคล้ายคืนค่าคู่ของระเบียบทุกคู่ระหว่างสองตารางที่คล้ายกัน ในบางสถานการณ์ การค้นหาดังกล่าวจะค้นหาด้วยข้อความที่คล้ายกันในช่วงเวลาเดียวกัน ข้อความเหล่านี้เรียกว่า ข้อความที่ใช้บ่อย จึงมีผู้ออกแบบการกรองโดยอิงกับข้อความที่ใช้บ่อยเพื่ออำนวยความสะดวกในการค้นข้อความชนิดนี้ ผู้ออกแบบใช้โครงสร้างเรียกว่าตารางค่าความคล้ายเพื่อตัดระเบียบที่ไม่เกี่ยวข้องออก การสร้างตารางค่าความคล้ายจะสร้างโดยอิงกับเซตของข้อความที่ใช้บ่อยที่ได้จากเซตของข้อความ อย่างไรก็ตาม ประสิทธิภาพของการกรองนี้ขึ้นอยู่กับเซตเหล่านี้โดยตรง เป้าหมายของวิทยานิพนธ์นี้คือเพื่อออกแบบวิธีค้นหาข้อความที่ใช้บ่อยที่เหมาะสมจะใช้กับการกรองโดยอิงกับข้อความที่ใช้บ่อย วิธีที่ออกแบบใช้ขั้นตอนการวิเคราะห์ที่จัดกลุ่มประเภทอิงกับความหนาแน่นชื่อ DBSCAN เพื่อจัดกลุ่มเซตของข้อความที่ใช้บ่อยที่คล้ายกันและค้นหาข้อความตัวแทนจากแต่ละกลุ่ม ผู้เขียนออกแบบขั้นตอนวิธีสองวิธี คือ DBRAN และ DBSM เพื่อกำจัดความซ้ำซ้อนของข้อความที่ใช้บ่อย DBRAN จัดกลุ่มข้อความที่ใช้บ่อยโดยใช้ DBSCAN และเลือกข้อความที่ใช้บ่อยอย่างสุ่มหนึ่งข้อความต่อหนึ่งกลุ่ม DBSM ใช้ DBSCAN เพื่อจัดกลุ่มเช่นกัน และผสมข้อความในกลุ่มจนกระทั่งไม่สามารถเพิ่มประสิทธิภาพการกรองได้ ขั้นตอนการวัดผลคือ นำวิธีที่ออกแบบไปใช้กับเซตของข้อความหลากหลายเซตสำหรับเซตของข้อมูลสามเซต และวัดประสิทธิภาพของการเชื่อมด้วยสายอักขระคล้ายที่ใช้การกรองชนิดนี้เป็นตัวกรอง พบว่า DBSM มีประสิทธิภาพดีกว่า DBRAN เมื่อแต่ละข้อความที่ใช้บ่อยคล้ายกันน้อย อย่างไรก็ตาม เมื่อข้อความที่ใช้บ่อยคล้ายกันมาก ประสิทธิภาพของทั้งสองวิธีใกล้เคียงกัน

ภาควิชา คณะนิเทศศาสตร์และวิทยาการ ลายมือชื่อนิสิต

 คอมพิวเตอร์ ลายมือชื่อ อ.ที่ปรึกษาหลัก

สาขาวิชา วิทยาการคอมพิวเตอร์และเทคโนโลยี

 สารสนเทศ

ปีการศึกษา 2557

5772601023 : MAJOR COMPUTER SCIENCE AND INFORMATION TECHNOLOGY

KEYWORDS: Similarity search; Similarity join; High-frequency queries; Cluster analysis; DBSCAN

KAMOLWAN KUNANUSONT: FINDING SETS OF HIGH-FREQUENCY QUERIES FOR HIGH-FREQUENCY-QUERY-BASED FILTER FOR SIMILARITY JOIN. ADVISOR: ASST. PROF. DR. JARULOJ CHONGSTITVATANA, 75 pp.

Similarity search and similarity join are important operations in text databases. Similarity search finds all records which are similar to the given text query while similarity join matches pairs of similar records from two relations. In some situations, some similar queries are repeated over a period of time. These queries are called high-frequency queries. High-frequency-query-based filter is used to facilitate this type of queries. This method uses an index structure called similarity table to prune non-related text records in relations. A similarity table is created based on a chosen high-frequency query obtained from the query set. However, the performance of this filter method depends mostly on these chosen queries. This thesis proposes a method to find high-frequency queries for the high-frequency-query-based filter. The proposed method is based on a density-based cluster analysis, called DBSCAN, to capture the main characteristics of the query set by grouping them and find the representative points from each group. Two methods – DBRAN and DBSM - to deal with redundant high-frequency queries are proposed. DBRAN finds clusters high-frequency queries, by DBSCAN, and randomly chooses one high-frequency query from a cluster as a representative. DBSM also uses DBSCAN to finds clusters, and repeatedly merge the queries in these clusters until it cannot give any improvement on similarity tables. For evaluation, the proposed method is applied on various sets of queries to find high-frequency queries for three datasets. It is found that DBSM performs better than DBRAN when the similarity between high-frequency queries is low. However, when the similarity between high-frequencies is high, the performance of both DBRAN and DBSM are about the same.

Department: Mathematics and Student's Signature

Computer Science Advisor's Signature

Field of Study: Computer Science and
Information Technology

Academic Year: 2014

ACKNOWLEDGEMENTS

I would like to express my very great appreciation to my advisor, Asst. Prof. Dr. Jaruloj Chongstitvatana, for her beneficial advice and patient suggestion on this thesis. My grateful thanks are also extended to Asst. Prof. Dr. Saranya Maneeroj whose question last year is the main inspiration of this work. Moreover, she also suggests some useful issues to improve the quality of my research.

Sincere thanks is offered to the external examiner, Dr. Kamol Keatruangkamala, whose advice provides improvements in various issues.

Special thanks should be given to Asst. Prof. Dr. Suphakant Phimoltares as his taught course contents provide me the idea of proposed method.

In addition, I would like to offer thanks to my junior student, Nathee Thitinanrunakit, as his guide advice and coding skill are very helpful.

Finally, I wish to thank my family and friends for their supports and sincere encouragement.

CONTENTS

	Page
THAI ABSTRACT	iv
ENGLISH ABSTRACT	v
ACKNOWLEDGEMENTS	vi
CONTENTS	vii
LIST OF FIGURES	ix
1. CHAPTER I INTRODUCTION	1
2. CHAPTER II RELATED WORKS	4
2.1 Similarity Join	4
2.1.1 Representation of Text in Text databases	4
2.1.2 Filter-and-verify Framework for Similarity Search and Similarity Join	7
- Prefix Filtering	8
- Positional Filtering	9
- Suffix Filtering	10
- Variable-length Prefix Scheme and AdaptJoin	11
- High-frequency-queries-based Filter	12
2.2 Cluster Analysis	20
2.2.1 Density-based Clustering Algorithm	20
- DBSCAN	21
2.2.2 Center-based Clustering Algorithm	22
- K-means	23
2.2.3 Hierarchical-based Clustering Algorithm	24
- Divisive Hierarchical-based Clustering Algorithm	24
- Agglomerative Hierarchical-based Clustering Algorithm	25
3. CHAPTER III PROPOSED METHOD	38
3.1 Definition of High-frequency queries.....	38

3.2 Methods to Find Sets of High-frequency queries.....	39
3.2.1 Brute Force Method	39
3.2.2 DBSCAN with Random Core Points (DBRAN)	42
3.2.3 DBSCAN with Merging Strategy (DBSM)	44
4. CHAPTER IV EXPERIMENT	49
4.1 Coverage Percentage	51
4.2 Numbers of In-coverage Candidates	54
4.3 Average Number of Candidates per In-coverage Query	56
5. CHAPTER V CONCLUSION	60
REFERENCES	xii
VITA	xiv



LIST OF FIGURES

	Page
Figure 1: Texts and their representative tokens in example 1.....	6
Figure 2: A similarity table for D_3	13
Figure 3: The visualization of the similar and dissimilar parts between texts	14
Figure 4: Visualization of figure 3 for maximum $f(d, F)$	15
Figure 5: Visualization of figure 3 for minimum $f(d, F)$	15
Figure 6: The range between upper bound and lower bound with different $f(q, F)$...	18
Figure 7: The example of the observable clusters.....	20
Figure 8: Example of core points, border points and noise points.....	22
Figure 9: K-means clustering results with different k	23
Figure 10: Divisive hierarchical-based clustering example.....	25
Figure 11: Initial data points of example 6.....	26
Figure 12: Initial proximity matrix of example 6.....	27
Figure 13: Data points of example 6 with 5 clusters.....	27
Figure 14: Proximity matrix of example 6 after first update	28
Figure 15: Data points of example 6 with 4 clusters.....	28
Figure 16: Proximity matrix of example 6 after the second update	29
Figure 17: Data points of example 6 with 3 clusters.....	29
Figure 18: Proximity matrix in example 6 after the third update	29
Figure 19: Data points in example 6 with two clusters	30
Figure 20: Proximity matrix in example 6 after the forth update.....	30
Figure 21: Final result of example 6.....	31
Figure 22: Initial data points of example 7.....	31

Figure 23: Initial proximity matrix of example 7.....	32
Figure 24: Data points of example 7 with 5 clusters.....	32
Figure 25: Proximity matrix of example 7 after first update.....	33
Figure 26: Data points of example 7 with 4 clusters.....	34
Figure 27: Proximity matrix of example 7 after the second update.....	34
Figure 28: Data points of example 7 with 3 clusters.....	35
Figure 29: Proximity matrix in example 7 after the third update.....	35
Figure 30: Data points in example 7 with two clusters.....	36
Figure 31: Proximity matrix in example 7 after the fourth update.....	36
Figure 32: Final result of Example 7.....	36
Figure 33: Visualizations of text group that brute force method cannot efficiently handle.....	41
Figure 34: Similarity tables and similarity between texts in example 8.....	47
Figure 35: Details of each dataset used in the experiment.....	49
Figure 36: Coverage percentage of DBLP with 60% related to original high-frequency queries compare with brute force method.....	52
Figure 37: Coverage percentage of DBLP with 60% related to original high-frequency queries.....	53
Figure 38: Coverage percentage of NYTimes with 60% related to original high-frequency queries.....	53
Figure 39: Coverage percentage of Enron with 60% related to original high-frequency queries.....	54
Figure 40: In-coverage candidates of DBLP with 60% related to original high-frequency queries.....	55

Figure 41: In-coverage candidates of NYTimes with 60% related to original high-frequency queries.....	56
Figure 42: In-coverage candidates of Enron with 60% related to original high-frequency queries.....	56
Figure 43: In-coverage candidates per query percentage of DBLP with 60% related to original high-frequency queries	57
Figure 44: In-coverage candidates per query percentage of NYTimes with 60% related to original high-frequency queries	58
Figure 45: In-coverage candidates per query percentage of Enron with 60% related to original high-frequency queries	58



CHAPTER I

INTRODUCTION

Similarity join is used to retrieve all similar pairs between two relations in a database. Texts pairs are considered to be similar if their similarity exceeds the specified threshold. The similarity between a pair of text is calculated by a similarity function. Many similarity functions such as Cosine similarity [1] and Jaccard similarity [1] are used. A brute force method for similarity join is to calculate the similarity between every pairs of text in the relations, and this takes long time to calculate. Many researches are done to reduce the time consumption. Filter-and-verify framework is a more efficient approach for similarity join. It eliminates some dissimilar pairs by comparing only some part of text and calculates the similarity of the remaining texts. Filter methods such as prefix filtering [2] are used in this framework.

Prefix filtering first divides each text string into two disjoint parts: prefix and suffix. The prefix part is used to evaluate the similar between pairs. If the pair of strings is considered as dissimilar from their prefixes, it is pruned. An efficient prefix filtering algorithm is AdaptJoin and AdaptSearch [3] which adaptively choose prefix length based on the performance of longer prefix compare to the previous one.

However, sometimes a set of queries contains the same patterns of texts. Prefix filtering, and most of filter method as well, do not take this type of queries in to account. Specifically, prefix filtering repeats the same filter process for the repeated queries or similar queries. If there are many such queries in the set, it consumes too much unnecessary time. [4] addresses the problem of the high-frequency queries, and proposes an index structure to support this issue called similarity table. This structure stores pointer of all texts string data according to their

similarity compared with the chosen high-frequency queries. This method can be applied with prefix filtering and solve the problem efficiently. However, an appropriate set of high-frequency queries need to be determined first.

An appropriate set of high-frequency queries are the set that covers all, or almost, of the repeated patterns in a set of queries while maintains the lowest space usage. Otherwise, when it is applied in high-frequency-queries-based filter, it will consume too much memory and become impractical. On the other hand, if the set is too small that is unable to capture the main characteristic of the query set, the time consumption of the filter method will be too high to accept. According to the constraint, the method to find the set of high-frequency queries should be designed in aware of both time and space complexity. To do this efficiency, the queries should be grouped together to capture all of their patterns first. The state-of-the-art method used to find the nature clusters of the dataset is cluster analysis [5].

Cluster analysis is an unsupervised method used in data mining, pattern recognition and other applications. This method aims to group data together based on the closeness between each data. Among three main types of cluster analysis are center based, hierarchical based and density based, the last one seem to be the best to applied in our problem. DBSCAN [6] is used to find sets of clusters in query set with the representative points. DBRAN randomly picks one core point from each cluster as a high-frequency query. However, these core points may not capture all main characteristics of the query set. The merging strategy is done to all core points to delete the redundancy as well as preserve the main characteristics of obtained set. The method of DBSCAN using merging strategy to find the sets of high-frequency queries is called DBSM.

DBSM and DBRAN are applied to find the high-frequency queries from the same query set. Then both sets are used in high-frequency-queries-based filter and the results are compared. The sets obtained from both methods are nearly the same with the low mutated query sets while DBSM is better if the query sets are highly mutated.



CHAPTER II

RELATED WORKS

This chapter describes the related works which contains two main parts: similarity join and cluster analysis.

2.1 Similarity Join

Given two text datasets D_1 and D_2 , a similarity function f and a similarity threshold t , the similarity join $SJ(D_1, D_2)$ is defined as:

$$SJ(D_1, D_2) = \{(d_1, d_2) \mid d_1 \in D_1 \wedge d_2 \in D_2 \wedge f(d_1, d_2) \geq t\}$$

The brute force method to find $SJ(D_1, D_2)$, which is to calculate $f(d_1, d_2)$ and compare with t for every possible pairs (d_1, d_2) in $D_1 \times D_2$, consumes too much time. Several methods are proposed to improve the efficiency of the algorithm for similarity join based on two representations of texts, which are described next.

2.1.1 Representation of Text in Text Databases

Texts can be represented in two different viewpoints. The first viewpoint is as the sequence of characters, called character-based viewpoint [2]. Another viewpoint is to map and consider each of them as a token, called token-based viewpoint [2].

- Character-based Viewpoint

Each text record in a dataset is treated as a text string. For instance, given a text 'System analysis needs various applications and software process', it is represent as the sequence of 'S', 'y', 's', 't', 'e', 'm', ' ', and so on. To count the number of overlapped strings between these two texts, we need to check whether each string pair contains the same character sequences or not. This procedure requires at least the length of the longer

string in the pair. For example, to check whether the string “system” and “systematic” in the second text are overlapped or not, we compare ‘s’ with ‘s’, ‘y’ with ‘y’, ‘s’ with ‘s’ and so on until found they are not the same string. We need to do this at least m times if m is the length of the longer text.

On the other hand, if two texts are mapped into a set of tokens instead, each string becomes an integer, which needs only one comparison to check the equality. To reduce the time of each comparison, tokens are used instead of texts as they require less time to compare. This is how texts are represented in token based viewpoint [2].

- Token-based Viewpoint

For each record d in a dataset D , d contains multiple text strings which can be mapped to tokens. These tokens are further used all along the computation instead of the original text strings. Example of this transformation is illustrated in example 1.

Example 1: Given a dataset D

$D = \{ \quad d_1: \{system, analysis, application, software, process\},$

$\quad d_2: \{computer, system, organization, processor\},$

$\quad d_3: \{image, processing, computer, vision, application\}$

Each text string can be mapped into token as figure 1

Text	Token
<i>System</i>	t_1
<i>Analysis</i>	t_2
<i>Application</i>	t_3
<i>Software</i>	t_4
<i>process, processor, processing</i>	t_5
<i>Computer</i>	t_6
<i>Organization</i>	t_7
<i>Image</i>	t_8
<i>Vision</i>	t_9

Figure 1: Texts and their representative tokens in example 1

D is transformed into the new dataset D'

$$D' = \{ \{t_1, t_2, t_3, t_4, t_5\},$$

$$\{t_1, t_5, t_6, t_7\},$$

$$\{t_3, t_5, t_6, t_8, t_9\} \}$$

String matching using character-based similarity considers *processor*, *process* and *processing* to be similar but not the same. However, they are of the same meaning, and could be considered the same in text queries. Token-based method overcomes this problem by mapping them into one token.

Although the token-based method is used, the time consumption is still high in practice. Filter and verify is one of the well-known framework to improve the performance.

2.1.2 Filter-and-verify Framework for Similarity Search and Similarity Join

To reduce the time for similarity join by brute-force method, filter-and-verify framework first *filters* out non candidates, which are text data which cannot possibly be answers of the query, and then *verifies* the remaining candidates for the answers. In the verification step, the similarity between the query and each candidate is calculated based on the similarity function, and only the candidate whose similarity exceeds the given threshold is returned as an answer.

Basically, what is done in the verification step is the same as the brute-force method. Thus, the performance of any similarity join in filter-and-verify framework depends on two factors. First, for each text data, filtering should need less time than verifying. Second, the filter step should produce only small number of candidates. The trade-off is necessary to minimize the filter time while maximizing the filter power.

Many methods for filtering are proposed. Some filtering methods minimize the filter time by examining only some parts of the text. Prefix filtering [2], positional filtering [7] and suffix filtering [7] are based on this idea. AdaptSearch and AdaptJoin [3] trade off the filter time and the filter power by increasing the parts of text data to be examined as long as it can increase the filter power. Another approach for filtering is to organize text data according to the similarity between the text data and a chosen text. For this approach, the filter power is high when the query is similar to the chosen text. For this reason, text which frequently appears in queries is chosen to organize all data. High-frequency-query based filter is an example of this approach. Next, these methods are further described.

- Prefix Filtering

Prefix filtering is a filter method which examines only the prefix of each text pair and finds out whether number of the common tokens between two texts exceeds the specific value or not before determines to keep or prune. If the record r contains s tokens, the value of needed overlap o is calculated from t from similarity function f , the l prefix is the first $s - o + l$ tokens of r . The pair (r_1, r_2) is pruned if the size of intersection between them is less than l . The following example illustrates the prefix filtering steps.

Example 2: Consider dataset D' in Example 1

$$D' = \{ \begin{array}{l} d_1: \{t_1, t_2, t_3, t_4, t_5\}, \\ d_2: \{t_1, t_2, t_6, t_7\}, \\ d_3: \{t_3, t_5, t_6, t_8, t_9\} \end{array} \}$$

and a query $q = \{t_1, t_2, t_6, t_7, t_8\}$ with the specified threshold 0.8 using cosine similarity as f and 2-prefix. First, the least overlap value to exceed threshold t is calculated, as shown below.

$$\frac{d_i \cap q}{\sqrt{|d_i||q|}} \geq t$$

$$d_i \cap q \geq 0.8 * \sqrt{5 * |d_i|}$$

$$o_i = \lceil 0.8 * \sqrt{5 * |d_i|} \rceil$$

Next, it is necessary to calculate o_i and 2-prefix of d_i , called pd_i , as well as the least overlap value of q that exceeds t , called o_q , and 2-prefix of q , called pd_q .

$$o_1 = 4, pd_1: \{t_1, t_2, t_3\}$$

$$o_2 = 4, pd_2: \{t_1, t_2, t_6\}$$

$$o_3 = 4, pd_3: \{t_3, t_5, t_6\}$$

$$o_q = 4, pq: \{t_1, t_2, t_6\}$$

After that, the number of common tokens, i.e. overlap part, between pq and each pd_i are computed, and compared with $l = 2$.

$$|pd_1 \cap pq| = 2 \geq 2, pd_1 \text{ is a candidate of } q.$$

$$|pd_2 \cap pq| = 3 \geq 2, pd_2 \text{ is a candidate of } q.$$

$$|pd_3 \cap pq| = 1 < 2, pd_3 \text{ is not a candidate of } q.$$

Finally, the cosine similarity of each candidate compare to q is calculated.

$$f(d_1, q) = 0.4 < 0.8$$

$$f(d_2, q) = 0.89 \geq 0.8$$

So, only d_2 is the answer of q .

- Positional Filtering

For very long text data, other filter method such as positional filtering and suffix filtering is used along with prefix filtering. Positional filtering considers the location of each token to estimate the highest possible similarity with another text string. The following example illustrates the positional filtering steps.

Example 3: Consider the dataset D_2

$$D_2 = \{ \begin{array}{l} d_1: \{t_1, t_2, t_3, t_4, t_5\}, \\ d_2: \{t_1, t_2, t_6, t_7\}, \\ d_3: \{t_3, t_5, t_6, t_8, t_9\} \end{array} \}$$

and a query $q = \{t_2, t_3, t_6, t_7, t_8\}$ with the specified threshold 0.8 using cosine similarity. First, find the least overlap value o that makes the cosine similarity exceeds the threshold t , using the same calculation as in example 2, we have

$$o = \lceil 0.8 * \sqrt{5 * |d_i|} \rceil$$

Therefore, o_1, o_2, o_3 and o_4 are 4. If the prefix filtering is used, d_2 would be one of the candidates as $pd_2: \{t_1, t_2, t_6\}$ and $pq: \{t_2, t_3, t_6\}$ (Assuming 2-prefix is used) since $|pd_2 \cap pq| = 2$, d_1 is a candidate of q . However, if the positional filtering is used, the maximum overlap between two text strings are considered as the sum of their prefix overlap and the number of remaining tokens of the shorter texts. That is, the maximum overlap is $|pd_2 \cap pq| + \min(1, 2) = 2 + 1 = 3$. As we need at least 4 overlap to satisfy 0.8 similarity threshold, d_2 cannot be the answer and will be pruned.

- Suffix Filtering

Although the positional filtering can prune some candidates that prefix filtering cannot, many non-related data still pass the filter. Suffix filtering improves the filter performance by also using suffix of each string to filter. The suffix is further divided into sub-prefix and sub-suffix and sub-positional filtering is used to prune more candidates. This procedure can be recursively applied until the remaining candidate size is small enough. Obviously, the more the suffix filtering is recursively applied, the more candidates can be

pruned while the longer the filter step takes. Therefore, trade-off between the cost of filter and verify must be considered for the overall performance.

Ppjoin+ [7] applies prefix filtering, positional filtering and suffix filtering, in this order. First the prefix filtering is applied, then, the positional filtering is applied with survived candidates, and finally the suffix filtering is then used. This method allows user to manually specify the number of times that suffix filtering is recursively called. Thus, the users can trade off the time and the number of candidates pruned.

- Variable-length Prefix Scheme and AdaptJoin

All of the mentioned filter method used ‘fixed-length prefix scheme’ in which the same prefix length is used for all strings. Specifically, the prefix length is fixed for all text data. The longer prefix length leads to more pruning power but also requires more time. However, there is no optimal prefix length for every string. J Wang, G. Li and J. Feng [3] suggests an efficient solution to this issue by adaptively choosing the length of prefix based on the information of estimated filter of prefix of length l in comparison with prefix of length $l + 1$. If the $(l + 1)$ prefix can eliminate enough strings to outweigh its filter cost, then the prefix length used in filtering is increased. This filter method is called AdaptJoin.

Although prefix filtering is an efficient method, it still repeats the same computation when the same query is repeated or similar queries appear. This type of queries, which is called high-frequency queries, need to be taken into account. Next, the high-frequency-queries-based filter [4], which is proposed to deal with this type of queries, is described.

- High-frequency-queries-based Filter

In some situations, the same or similar texts are frequently queried in the database. For example, after April 2015 Nepal earthquake [8], the search for words ‘Nepal’, ‘earthquake’ and ‘help Nepal’ dramatically increases compared with the previously recorded data by Google trends explorer [9]. These groups of words are ‘high-frequency queries’. Therefore, a set high-frequency query in the query set Q is defined as a query such that many other queries in the same set are similar with it.

If such query texts or the similar query texts are repeated often in a period of time, prefix filtering repeatedly processes the same or similar steps for every query, which is not efficient. [4] takes this into account and proposes an index structure called similarity table. This table keeps pointers to all texts in the data set sorted according to the similarity with the chosen high-frequency query.

○ Similarity Table

When high-frequency queries are chosen, a similarity table is created based on each of them. Each table stores the pointers to all records in the dataset sorted by the similarity between each record and the corresponding high-frequency query. For the high-frequency query F of dataset D , the row i of s rows similarity table ST_F stores the pointers to the data records which are similar to F , with the similarity value between i/s and $(i+1)/s$. Therefore, for each row $ST_F[i]$ of ST_F :

$$ST_F[i] = \{p \mid p \text{ is the pointer to } r \in D, i/s < f(r, F) \leq (i+1)/s \} .$$

The following example illustrates the similarity table.

Example 4: Consider a dataset D_3

$$D_3 = \{ d_1: \{t_1, t_2, t_3, t_4, t_5\},$$

$$d_2: \{t_1, t_2, t_6, t_7\},$$

$$d_3: \{t_2, t_3, t_4\},$$

$$d_4: \{t_3, t_4, t_5, t_8, t_9\} \}$$

Suppose $F = \{t_1, t_3, t_4, t_5, t_9\}$ and f is jaccard similarity function, where

$$f(d_1, d_2) = \frac{|d_1 \cap d_2|}{|d_1 \cup d_2|}. \text{ We calculate } f(d_i, F) \text{ for all } 1 \leq i \leq 4.$$

$$f(d_1, F) = 3/(10-3) = 0.43$$

$$f(d_2, F) = 2/(9-2) = 0.29$$

$$f(d_3, F) = 2/(8-2) = 0.33$$

$$f(d_4, F) = 4/(10-4) = 0.67$$

Next, the similarity table with 4 rows is constructed, p_i denotes the pointer to d_i :

Row	Jaccard similarity (0.0-1.0]	Records
0	(0.0, 0.25]	-
1	(0.25, 0.5]	d_1, d_2, d_3
2	(0.5, 0.75]	d_4
3	(0.75, 1.0]	-

Figure 2: A similarity table for D_3

The similarity between a candidate text data and the query can be estimated from the similarity between the query and the high-frequency query and the similarity between the text data and the high-frequency query. That means, any candidate answers of a query can be in only some specific rows of the similarity table. The high-frequency-queries-based filter method

filters out some non-related candidates based on this idea. Given a high-frequency query F and a new query q with threshold t , this filter method finds the upper and the lower bounds of the similarity between q and any text data d in the dataset from the known similarity between F and d .

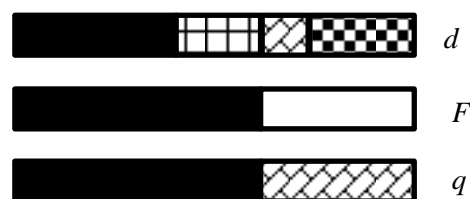







Figure 3: The visualization of the similar and dissimilar parts between texts

Consider figure 3, five box patterns indicate different kinds of similarity of each text record.

- The black area  indicates the sharing tokens that lead to the similarity value of three text records d , F and q .
- The white area  indicates tokens that are in F but not in q . This lead to the dissimilarity between F and q .
- The large-grid-pattern area  indicates the area that can be similar with both F and q .
- The diagonal-brick-pattern area  indicates the sharing tokens between d and q but not in F .
- The checker-pattern area  indicates the tokens in d that are not in both F and q .

When q is queried, $f(q, F)$ is calculated. Only the black areas of F and q , the white area and the diagonal-brick-pattern area are known. All of the areas in d are different according to their similarity with F . The unknown similarity

area can be estimated from $f(q, F)$. In the case, we want to find two values: the highest and lowest value of $f(d, F)$ that $f(q, d)$ still exceeds t . Two different scenarios are needed to be considered.

1. Maximum $f(d, F)$






This scenario happens when $f(q, F) \leq f(d, F)$ and the  area becomes  in figure 3 as shown in figure 4




Figure 4: Visualization of figure 3 for maximum $f(d, F)$

In this case, the similarity between d and q is the sum of  area and  in d . That is, if $f(q, F) < t$. We need at least $t - f(q, F)$ of  area. In the similarity table, this is the row that covers the similarity value $1 - (t - f(q, F))$. So, the *upper bound* or the highest possible value of $f(d, F)$ can be calculated as shown below.

$$\begin{aligned} \text{Upper bound: } f(d, F) &= 1 - (t - f(q, F)) \\ &= f(q, F) + (1 - t) \end{aligned}$$

2. Minimum $f(d, F)$

This scenario happens when all tokens in q but not in F are in d . This means most of $f(d, q)$ results from the  area. This scenario is shown in figure 5.

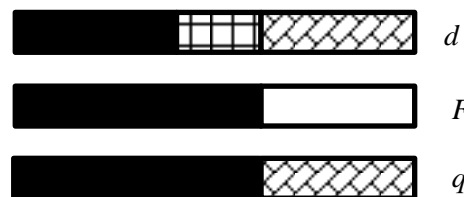




Figure 5: Visualization of figure 3 for minimum $f(d, F)$.

We want the sum of the similarity of  and  areas of d to exceed t . That is, $f(d, F) + (1 - f(q, F)) \geq t$. If we rearrange the equation, we have the *lower bound* or the lowest possible value of $f(d, F)$ can be calculated as shown below.

$$\begin{aligned} \text{Lower bound:} \quad f(d, F) &= t - 1 + f(q, F) \\ &= f(q, F) - (1 - t) \end{aligned}$$

High-frequency-queries-based filter ensures that for the new query q , the candidates are in the rows in the range $[f(q, F) - (1 - t), f(q, F) + (1 - t)]$ of the similarity table ST_F . Therefore, for a high-frequency query F and a text query q with threshold t , the candidates obtained from high-frequency-queries-based filter, denoted by $HFQB(F, q)$, is defined as follows.

$$HFQB(F, q) = \{d \mid d \in ST_F[i], \text{ for all } i \text{ such that } f(q, F) - (1 - t) * s < i \leq f(q, F) + (1 - t) * s, s \text{ is the size of } ST_F\}$$

Assume that the new query q is queried with threshold t , high-frequency-queries-based filter would be processed as follows:

- For each F in the high-frequency query set, compute $f(q, F)$
- Find F that $f(q, F)$ is highest
- Compute $f(q, F) - (1 - t)$ and $f(q, F) + (1 - t)$.
- For each row $i \in [[s * f(q, F) - (1 - t)], [s * f(q, F) + (1 - t)]]$
 - For each text record d in row i , add d as to the candidate set of q .
- For each text record d in the candidate set of q , compute the real $f(d, q)$.
If $f(d, q) \geq t$, add d to the set of answer of q .
- Return the answer set.

Without loss of generality, the filter step of one similarity table is shown in Example 5.

Example 5: Suppose the query $q = \{t_1, t_3, t_4, t_5, t_{10}\}$ is queried with threshold = 0.95 with the similarity table shown in figure 2 with $F = \{t_1, t_3, t_4, t_5, t_9\}$. Since ST_F is created based on jaccard similarity, $f(q, F)$ is computed using jaccard similarity. Therefore, $f(q, F) = 4/(10-4) = 0.67$. The lower bound is $4(0.67-(1-0.95)) = 2.52$ which is row 2. The upper bound is $4*(0.67+(1-0.95)) = 2.88$ which is also row 2. The candidate for q is only $d_4: \{t_3, t_4, t_5, t_8, t_9\}$. Then d_4 is verified by calculate $f(q, d_4) = 3/(10-3) = 0.43$. This means d_4 is not the answer of q .*

The query q in Example 5 is not quite similar with F but the threshold t is high. Therefore the answer range is restricted at only one row. In practical, the query q and F can be dissimilar, says $f(q, F) < 0.5$, and the threshold can be low. Since the equation to find the upper bound and the lower bound from the similarity table is $f(q, F) \pm (1-t)$, we consider two parameters separately:

- The Similarity Value of the Query Text and the High-frequency Query $f(q, F)$

The row i that covers $f(q, F)$ of ST_F is the row that always contains the candidates. If the threshold $t = 1$, this row is both the upper and the lower bound. Otherwise, the answer range spreads up $s*(1-t)$ rows above and below from this row as shown in figure 6. Therefore, if $f(q, F)$ is higher, the upper bound may reaches the topmost row, which means that the number of rows that can contain the answers is lower.

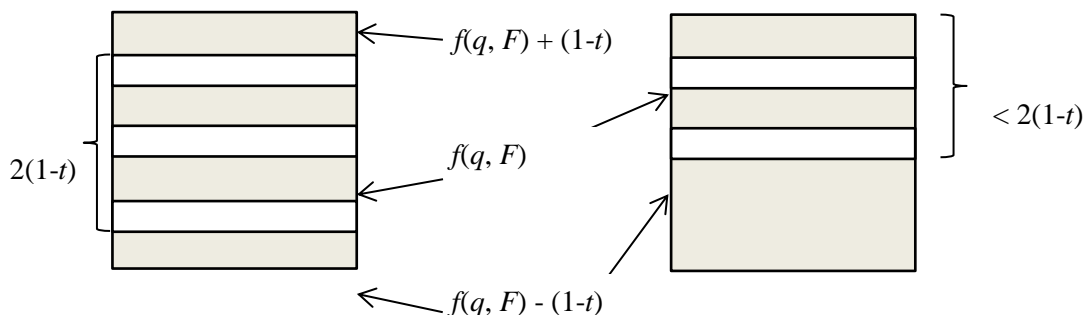


Figure 6: The range between upper bound and lower bound with different $f(q, F)$

It is possible that if $f(q, F)$ is lower, the lower bound can reach the bottom row. This scenario seems similar to the previous scenario and should result in better performance of the filter as the number of rows to be checked is fewer. However, most of the records in the dataset are not and should not be similar with the high-frequency queries. The reason is that if most records are similar with F , ST_F may not be able to filter out the non-related candidates for the given query if the query is similar with F which is unreasonable. Therefore, most of the similarity tables contain a huge number of pointers at the lower rows. Hence, the higher $f(q, F)$ is more preferable.

Although the effect of $f(q, F)$ to the performance of high-frequency-queries-based filter is significant, the similarity threshold t can have more impact. This is described in the next section.

- Similarity Threshold t

As mentioned earlier, the span-out of the answer rows is equal to $2*(1-t)$. If the value of t is lower, more rows are needed to be verified. If $f(q, F) = 0.9$ but the similarity threshold is very low, says 0.2, the lower bound = 0.1 and the upper bound is 1 (The exact value is 1.7, but the similarity value is at most 1.0). This means almost all dataset records are the candidates

although q and F are very similar. Normally, the similarity threshold is at least 0.5 in most similarity join and similarity search researches.

Both $f(q, F)$ and the similarity threshold t have crucial effect on the performance of high-frequency-queries-based filter. Since the similarity threshold t is specified by users, no change can be done. On the other hand, the improvement regarding $f(q, F)$ is possible. If the set of high-frequency queries can be chosen so that for every new query q , there is at least an F in the set that q and F is similar, $f(q, F)$ would be high for every new query q and the filter power can be improved. Hence, appropriate set of high-frequency queries are necessary to create the efficient similarity table for similarity join and search. To find such the set, we need to have the set of previous queries to analyze its characteristics. The query set may or may not contain high-frequency queries and this is undesirable. Furthermore, if there are any high-frequency queries, their number remains unknown until the set is analyzed with some method.

This thesis focuses on the problem to find an appropriate set of high-frequency queries. Based on high-frequency-queries-based filter method, the properties of the desired set of high-frequency queries can be described as follows.

- a. For most of the query in the query set, at least one of the members of this set must be similar with it.
- b. Each member should not similar with each other because it can lead to the redundancy in the set.
- c. The set should large enough to pass the a. constraint but not too large to violate the b. constraint.

Due to three constraints of the appropriate set of high-frequency queries, the information about the group of queries in the query set is needed. Queries that are similar should be grouped together and find the representative texts of the group to be the high-frequency queries. The tasks to find groups in the dataset are called cluster analysis.

2.2 Cluster Analysis

Cluster analysis [5] is an unsupervised task to find groups of data in the dataset. It is used in many fields such as data mining, machine learning and pattern recognition. Clustering algorithms, which are used in cluster analysis, has three main types: center-based, hierarchical-based and density-based.

2.2.1 Density-based Clustering Algorithm

Density-based clustering groups the data in the high density areas separated by lower density areas. Consider figure 7 as the examples:

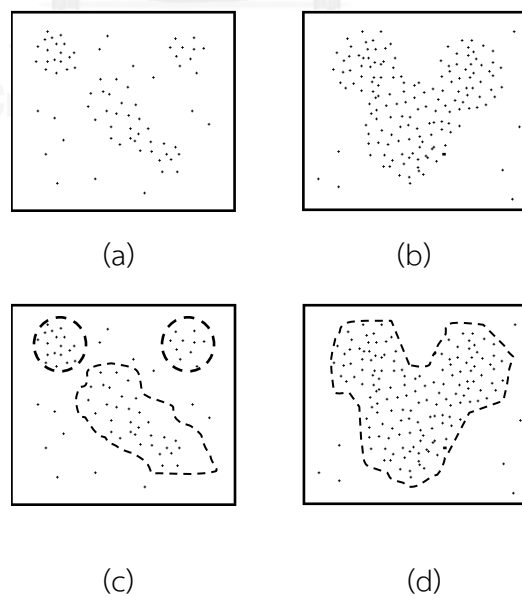


Figure 7: The example of the observable clusters

In figure 7, clusters from both (a) and (b) are observable as shown inside the dash lines in (c) and (d) respectively. This is because the natural clusters are the areas that the points are compactly located. Although there are some points that are located away from other points, they are not classified as clusters. This is one strategy to define 'density' in density-based cluster analysis. The main advantage of this definition is that clusters with the non-globular or non-structure shape can be handled efficiently. The method which is based on this density definition is DBSCAN.

- DBSCAN

DBSCAN [6] is the density-based clustering algorithm that finds the clusters by eliminates some points in the low density areas while group those in higher density areas together. Unlike another clustering algorithm, DBSCAN major advantage is that it can find clusters correctly although the data are of many noise points. Groups of points are classified as the clusters if there are enough numbers of points that located near each other. The minimum number of points *minpt* and the radius of clusters *eps* are two parameters which need to be chosen for DBSCAN. Three types of data points separated by DBSCAN are core point, border point and noise point:

- Core points are the points that contains at least than *minpt* points in distance *eps*.
- Border points are points within *eps* distance from one of core points but not core points.
- Noise points are other points which are neither core nor border points.

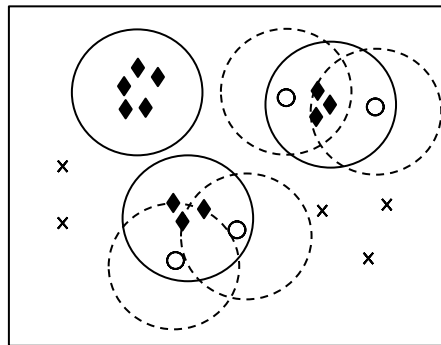


Figure 8: Example of core points, border points and noise points

Let $minpt = 4$ and eps be the radius of clusters. In figure 8, diamond-shaped objects symbolize core points, while small-white-circles represent border points and cross-shaped objects represent the noise points.

Although DBSCAN is firstly proposed to support spatial database [6], it can be applied with another applications efficiently. Similarity functions can also be used instead of distance function in order to define the relationship between data, as state by [10]. In this paper, similarity function for similarity join and search is used. Regardless of the functions used, the core points of the clusters are also of the same meaning. That is, they are the points with many points located not far from, in another word similar to, them. If the DBSCAN is applied with similarity join queries set, the appropriate set of high-frequency queries should be among the core points. Based on this assumption, the algorithm to find the high-frequency queries set from the query set is proposed and described next chapter.

2.2.2 Center-based clustering algorithm

This type of clustering algorithms is used to find clusters such that each data point belongs to the cluster whose center is nearest to the point. K-means algorithm is an example of center-based clustering.

- K-means algorithm [5]

K-means chooses centers of all clusters randomly, assigns every data in a cluster with the closest center and re-computes the centers until stable. Given a dataset D , the similarity function (or distance function) f and a number k . Cluster analysis by K-means can be done by the following steps:

- 1) Pick k data points as the center (can be done by randomly or another better procedure).
- 2) For each of the rest data d and each center c , compute $f(c, d)$ and assign d in the same group which $f(c, d)$ is closest.
- 3) For each cluster obtained in 2), recomputed the new center c .
- 4) Repeat 2) and 3) until the center not change.

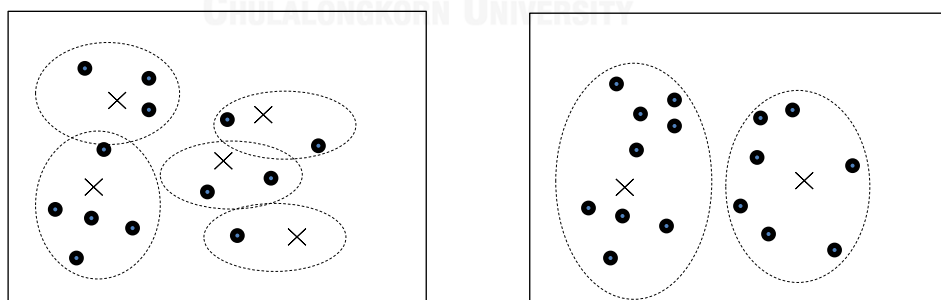


Figure 9: K-means clustering results with different k

One main disadvantage of K-means algorithm is that the number of clusters is needed. Consider figure 9, the different value of k leads to different clusters retrieved from the algorithm. [11] modified K-means by using singular

vector decomposition algorithm to choose the value of k before apply the cluster analysis. Another disadvantage of K-means is that the result depends mostly on the initial location of the center-point. Many researches such as [12] and [13] provide the solution to this issue.

K-means can be applied to find the appropriate set of high-frequency queries by using the best set of c as the high-frequency queries set. However, due to the disadvantage of center initial location and the unknown variable k , it cannot be used immediately.

2.2.3 Hierarchical-based Clustering Algorithm

Hierarchical-based clustering algorithm groups the data together by their hierarchical relationship. Specifically, the data points which are located close together have stronger relationship and more likely to be in the same clusters than the data points located further. Therefore, if a cluster is needed to be split, the close points will be in the same cluster while the further points located in another cluster. Similarly, if two clusters are needed to be merged, there are two types of this clustering algorithm. The first one, which is based on top-down strategy, is divisive hierarchical-based clustering algorithm.

- Divisive Hierarchical-based Clustering Algorithm

This method starts with one cluster of all data in the dataset. Then try to split a cluster into two clusters. Then choose one of these two clusters and further split it into another two clusters, to produce three clusters. This procedure is repeated until each data point became a single cluster of itself.

The performance of this method depends on two factors. To determine which cluster is the most appropriate to split and how to split it. Mostly, a cluster is chosen to be split first if the distribution of its data points is the least compact. After the cluster is chosen to be split, Flat algorithm [14] is applied to determine the best way to divide. Figure 10 shows the example of divisive hierarchical based clustering algorithm.

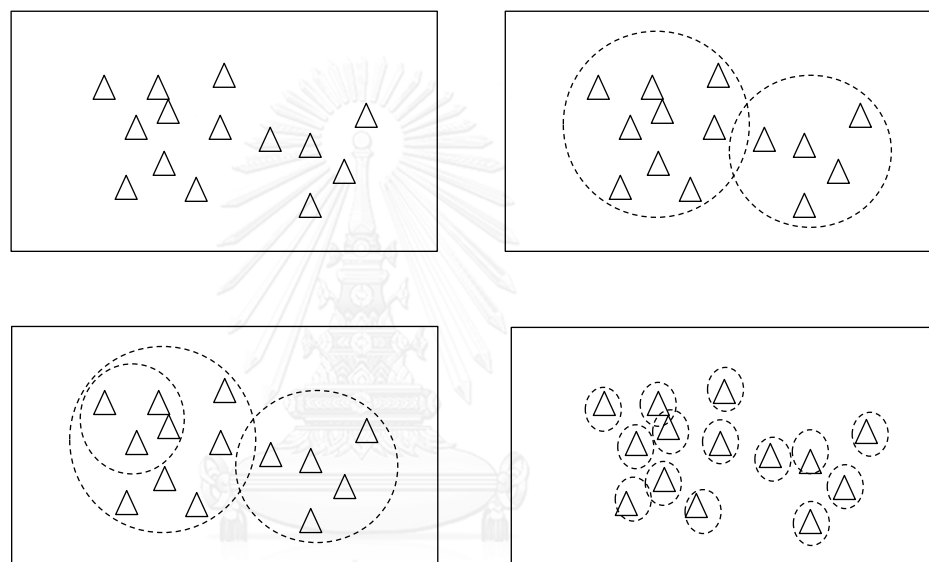


Figure 10: Divisive hierarchical-based clustering example

- Agglomerative Hierarchical-based Clustering Algorithm

In contrast to divisive hierarchical-based clustering algorithms, agglomerative method starts with each single data point and tries to group it with the nearest (or most similar) point to form a cluster. This procedure is repeated until all becomes only one cluster. Many techniques are applied to choose the most appropriate pair to be merged.

Proximity matrix is the matrix that stores the information about the similarity or distance between the clusters. The values in the

matrix are used to determine the next pair to be merged. After each merge occurs, the value inside each row and column corresponds to the clusters are needed to be recomputed based on the matrices to measure the cluster distance. Some of the widely-used methods are described:

- MIN or Single Link Method [5]

Suppose there are k clusters, and the distance between two clusters is defined by the minimum distance between the data points between the two clusters. Two clusters that have the lowest distance between them should be merged together and the distance between this new cluster and another $k-2$ clusters are needed to be recomputed.

Example 6: Suppose we have six points in 2-dimensional space: $(0.7, 2.7)$, $(1.8, 3.2)$, $(2.6, 0.8)$, $(1.5, 2.3)$, $(3.0, 1.2)$, $(2.2, 1.5)$. The scatter plot of these points is shown below in figure 11. If the single link method for agglomerative hierarchical-based clustering is used to group the data and the distance is measured by Euclidean distance. First the proximity matrix is constructed, shown in figure 12.

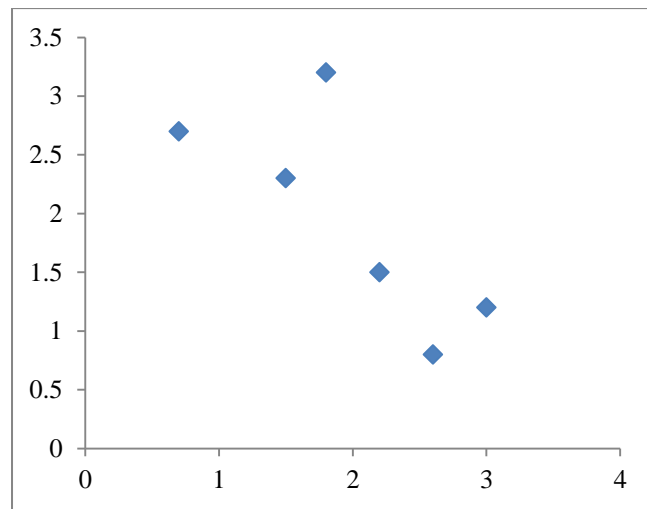


Figure 11: Initial data points of example 6

	(0.7, 2.7)	(1.8, 3.2)	(2.6, 0.8)	(1.5, 2.3)	(3.0, 1.2)	(2.2, 1.5)
(0.7, 2.7)	0	1.21	2.69	0.89	2.75	1.92
(1.8, 3.2)	1.21	0	2.53	0.95	2.33	1.75
(2.6, 0.8)	2.69	2.53	0	1.86	0.57	0.81
(1.5, 2.3)	0.89	0.95	1.86	0	1.86	1.06
(3.0, 1.2)	2.75	2.33	0.57	1.86	0	0.85
(2.2, 1.5)	1.92	1.75	0.81	1.06	0.85	0

Figure 12: Initial proximity matrix of example 6

The minimum distance value is 0.57 which is between (2.6, 0.8) and (3.0, 1.2). Therefore these two points are merged into same clusters. The merging result is shown in figure 13.

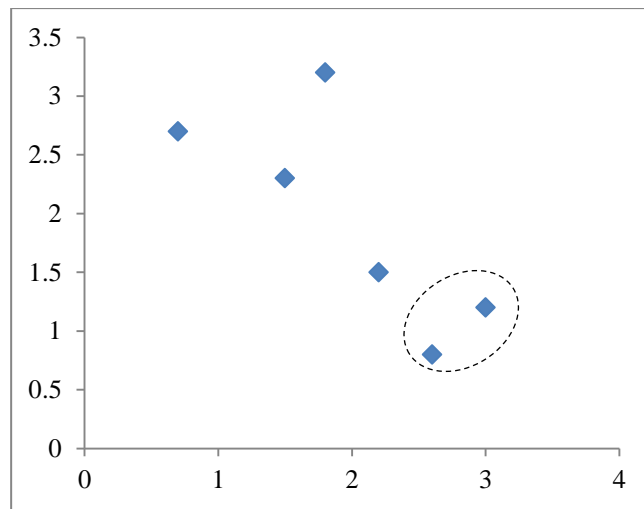


Figure 13: Data points of example 6 with 5 clusters

Now, to update the value in the proximity matrix, first the columns and rows related to these two points are merged into one column and row respectively. The value in each new cell is selected from the minimum value between two cells regards these two points. For example, the distance between $(2.6, 0.8)$, $(3.0, 1.2)$ and $(0.7, 2.7)$ is 2.69 as $2.69 < 2.75$. The proximity matrix is shown in figure 14.

	(0.7, 2.7)	(1.8, 3.2)	(2.6, 0.8), (3.0, 1.2)	(1.5, 2.3)	(2.2, 1.5)
(0.7, 2.7)	0	1.21	2.69	0.89	1.92
(1.8, 3.2)	1.21	0	2.33	0.95	1.75
(2.6, 0.8), (3.0, 1.2)	2.69	2.33	0	1.86	0.81
(1.5, 2.3)	0.89	0.95	1.86	0	1.06
(2.2, 1.5)	1.92	1.75	0.81	1.06	0

Figure 14: Proximity matrix of example 6 after first update

Next, the minimum value is 0.81, which is the distance between $\{(2.6, 0.8), (3.0, 1.2)\}$ and $(2.2, 1.5)$. The point $(2.2,$

1.5) is merged with the cluster as shown in figure 15 and the updated proximity matrix is shown in figure 16.

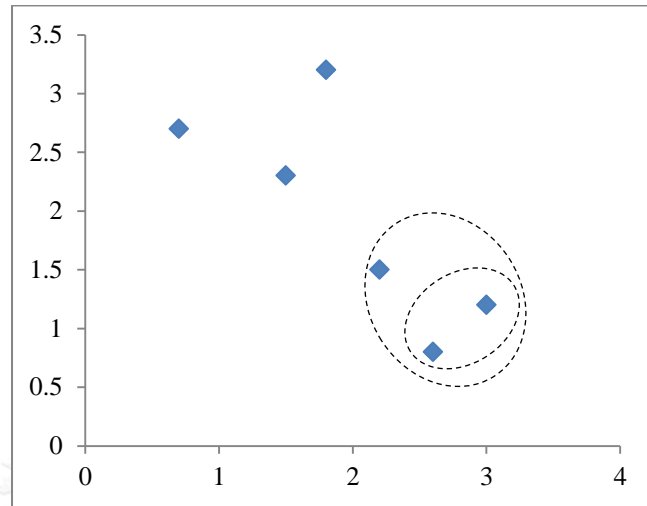


Figure 15: Data points of example 6 with 4 clusters

	(0.7, 2.7)	(1.8, 3.2)	(2.6, 0.8), (3.0, 1.2), (2.2, 1.5)	(1.5, 2.3)
(0.7, 2.7)	0	1.21	1.92	0.89
(1.8, 3.2)	1.21	0	1.75	0.95
(2.6, 0.8), (3.0, 1.2), (2.2, 1.5)	1.92	1.75	0	1.06
(1.5, 2.3)	0.89	0.95	1.06	0

Figure 16: Proximity matrix of example 6 after the second update

After that, the minimum distance in the proximity matrix is 0.89. The points (0.7, 2.7) and (1.5, 2.3) are merged to create a new cluster. Figure 17 shows the 3-cluster result and the corresponding updated proximity matrix is shown in figure 18.

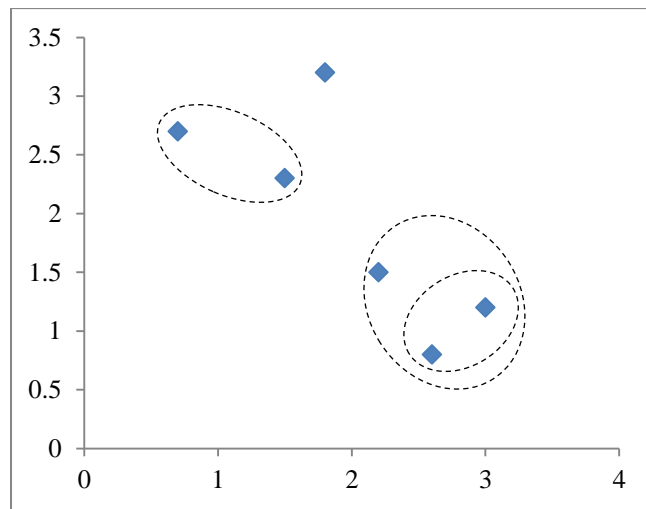


Figure 17: Data points of example 6 with 3 clusters

	(0.7, 2.7), (1.5, 2.3)	(1.8, 3.2)	(2.6, 0.8), (3.0, 1.2), (2.2, 1.5)
(0.7, 2.7) (1.5, 2.3)	0	0.95	1.06
(1.8, 3.2)	0.95	0	1.75
(2.6, 0.8), (3.0, 1.2), (2.2, 1.5)	1.06	1.75	0

Figure 18: Proximity matrix in example 6 after the third update

The minimum value in the proximity matrix is 0.95. The point (1.8, 3.2) is merged with the cluster {(0.7, 2.7), (1.5, 2.3)}. The results and proximity matrix are shown in figure 19 and 20 respectively.

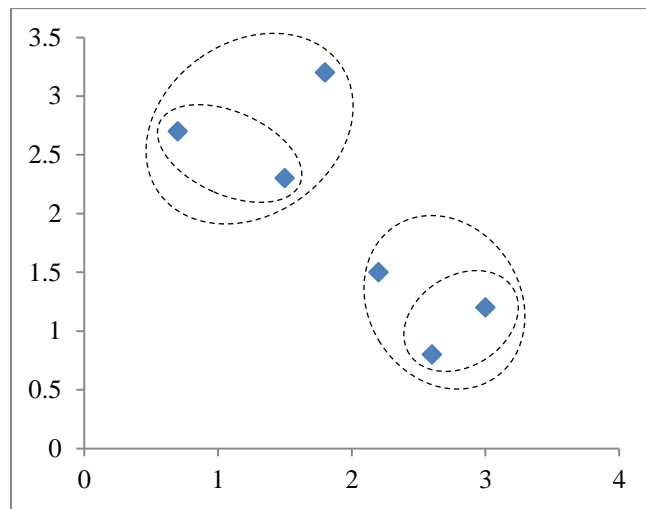


Figure 19: Data points in example 6 with two clusters

	(0.7, 2.7), (1.5, 2.3), (1.8, 3.2)	(2.6, 0.8), (3.0, 1.2), (2.2, 1.5)
(0.7, 2.7) (1.5, 2.3), (1.8, 3.2)	0	0.95
(2.6, 0.8), (3.0, 1.2), (2.2, 1.5)	0.95	0

Figure 20: Proximity matrix in example 6 after the fourth update

Finally, two clusters are merged into one cluster as in figure 21.

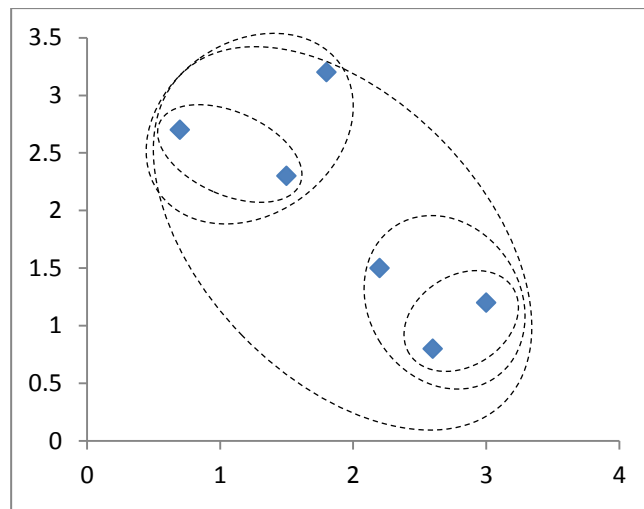


Figure 21: Final result of example 6

○ MAX or Complete Link Method [5]

In contrast to the single link method, the distance between two clusters is defined by the maximum distance between the data points between the two clusters. The reason behind this is that sometimes the single link method merges the long clusters which the nearest points are close but the furthest points distance is very high.

Consider the dataset in Example 6.

Example 7: Suppose we have six points in 2-dimensional space: $(0.7, 2.7)$, $(1.8, 3.2)$, $(2.6, 0.8)$, $(1.5, 2.3)$, $(3.0, 1.2)$, $(2.2, 1.5)$. The scatter plot of these points is shown below in figure 22. If the complete link method for agglomerative hierarchical-based clustering is used to group the data and the distance is measured by Euclidean distance. First the proximity matrix is constructed as shown in figure 23.

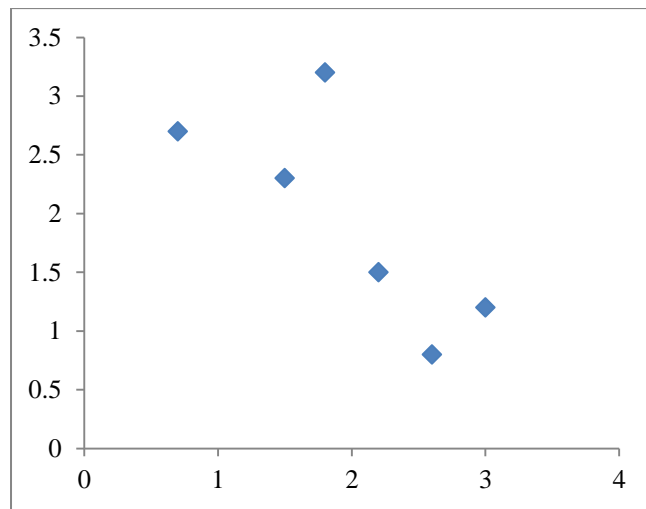


Figure 22: Initial data points of example 7

	(0.7, 2.7)	(1.8, 3.2)	(2.6, 0.8)	(1.5, 2.3)	(3.0, 1.2)	(2.2, 1.5)
(0.7, 2.7)	0	1.21	2.69	0.89	2.75	1.92
(1.8, 3.2)	1.21	0	2.53	0.95	2.33	1.75
(2.6, 0.8)	2.69	2.53	0	1.86	0.57	0.81
(1.5, 2.3)	0.89	0.95	1.86	0	1.86	1.06
(3.0, 1.2)	2.75	2.33	0.57	1.86	0	0.85
(2.2, 1.5)	1.92	1.75	0.81	1.06	0.85	0

Figure 23: Initial proximity matrix of example 7

The minimum distance value is 0.57 which is between (2.6, 0.8) and (3.0, 1.2). Therefore these two points are merged into same clusters as in figure 24.

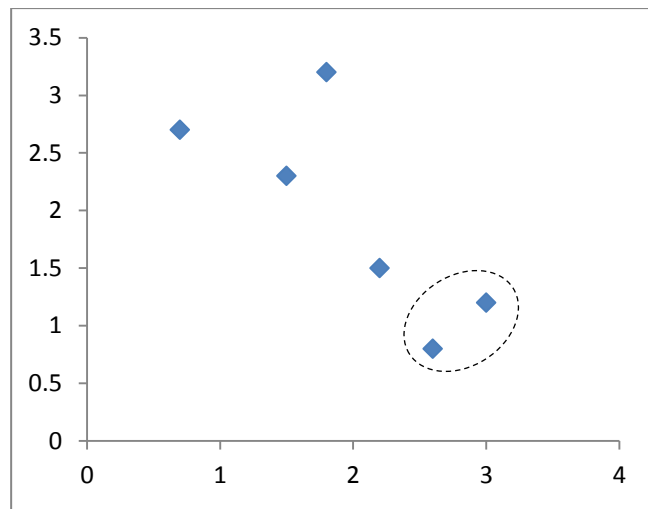


Figure 24: Data points of example 7 with 5 clusters

Now, to update the value in the proximity matrix, first the columns and rows related to these two points are merged into one column and row respectively. The value in each new cell is selected from the maximum value between two cells regards these two points. For example, the distance between $(2.6, 0.8)$, $(3.0, 1.2)$ and $(0.7, 2.7)$ is 2.75 as $2.69 < 2.75$. The proximity matrix after this step is shown in figure 25.

	$(0.7, 2.7)$	$(1.8, 3.2)$	$(2.6, 0.8), (3.0, 1.2)$	$(1.5, 2.3)$	$(2.2, 1.5)$
$(0.7, 2.7)$	0	1.21	2.75	0.89	1.92
$(1.8, 3.2)$	1.21	0	2.53	0.95	1.75
$(2.6, 0.8), (3.0, 1.2)$	2.75	2.53	0	1.86	0.85
$(1.5, 2.3)$	0.89	0.95	1.86	0	1.06
$(2.2, 1.5)$	1.92	1.75	0.85	1.06	0

Figure 25: Proximity matrix of example 7 after first update

Next, the minimum value is 0.85, which is the distance between $\{(2.6, 0.8), (3.0, 1.2)\}$ and $(2.2, 1.5)$. The point $(2.2, 1.5)$ is merged with the cluster as in figure 26 and the proximity matrix is updated in figure 27.

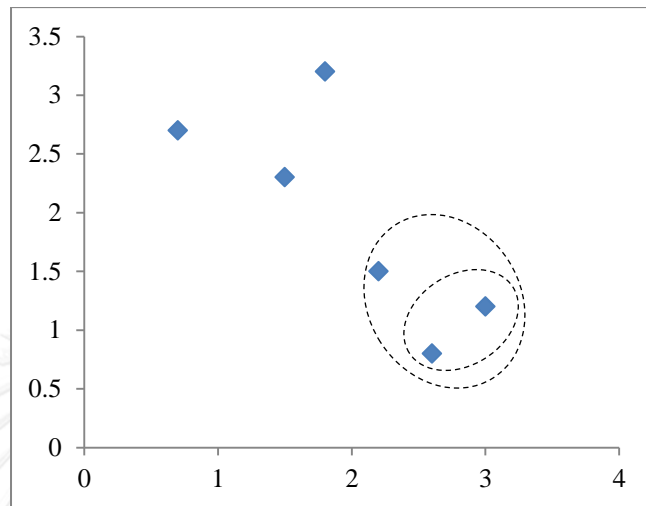


Figure 26: Data points of example 7 with 4 clusters

	(0.7, 2.7)	(1.8, 3.2)	(2.6, 0.8), (3.0, 1.2), (2.2, 1.5)	(1.5, 2.3)
(0.7, 2.7)	0	1.21	2.75	0.89
(1.8, 3.2)	1.21	0	2.53	0.95
(2.6, 0.8), (3.0, 1.2), (2.2, 1.5)	2.75	2.53	0	1.86
(1.5, 2.3)	0.89	0.95	1.86	0

Figure 27: Proximity matrix of example 7 after the second update

After that, the minimum distance in the proximity matrix is 0.89. The points $(0.7, 2.7)$ and $(1.5, 2.3)$ are merged to create a new cluster. Figure 28 and 29 show the data point clusters and the proximity matrix respectively.

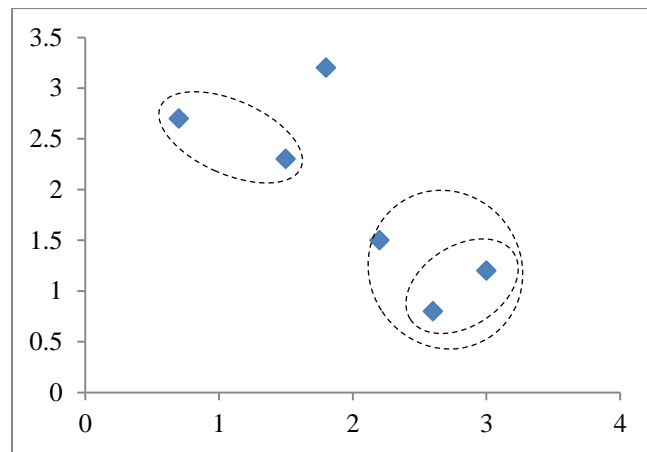


Figure 28: Data points of example 7 with 3 clusters

	(0.7, 2.7), (1.5, 2.3)	(1.8, 3.2)	(2.6, 0.8), (3.0, 1.2), (2.2, 1.5)
(0.7, 2.7) (1.5, 2.3)	0	1.21	2.75
(1.8, 3.2)	1.21	0	1.75
(2.6, 0.8), (3.0, 1.2), (2.2, 1.5)	2.75	1.75	0

Figure 29: Proximity matrix in example 7 after the third update

The minimum value in the proximity matrix is 1.21. The point (1.8, 3.2) is merged with the cluster $\{(0.7, 2.7), (1.5, 2.3)\}$. The result is shown in figure 30 while the updated proximity matrix is in figure 31.

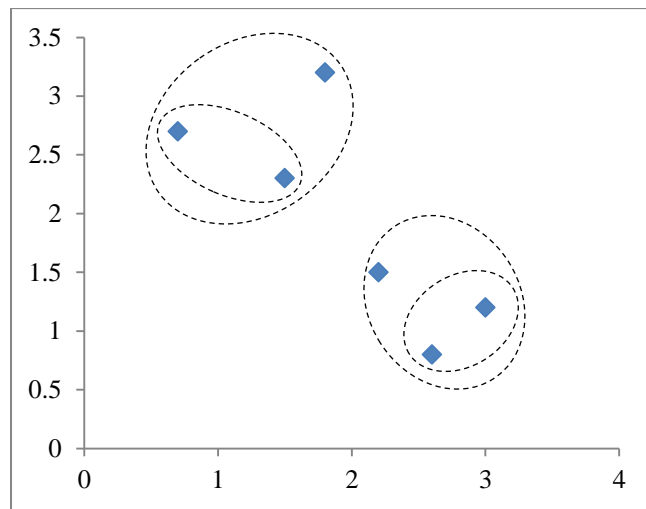


Figure 30: Data points in example 7 with two clusters

	$(0.7, 2.7), (1.5, 2.3), (1.8, 3.2)$	$(2.6, 0.8), (3.0, 1.2), (2.2, 1.5)$
$(0.7, 2.7), (1.5, 2.3), (1.8, 3.2)$	0	2.75
$(2.6, 0.8), (3.0, 1.2), (2.2, 1.5)$	2.75	0

Figure 31: Proximity matrix in example 7 after the fourth update

Finally, two clusters are merged into one cluster as in figure 32.

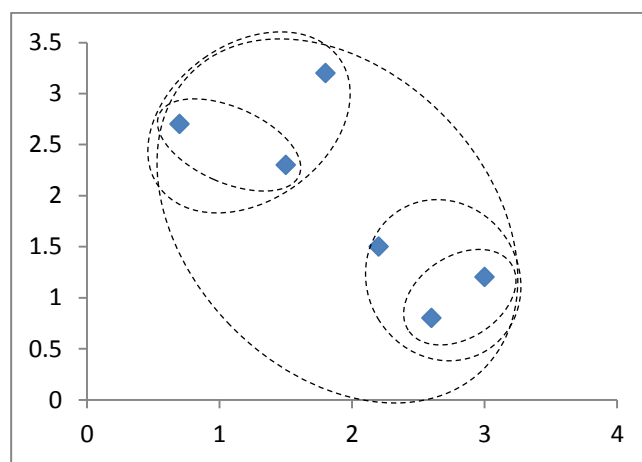


Figure 32: Final result of example 7

In this example, the order which the points are merged is the same for both single link and complete link but the value in the proximity matrix is different. There is some situations that this different leads to the different order or merging since the criteria to select the next pair to merge depends on the minimum value.

Hierarchical-based clustering method is not appropriate to find the set of high-frequency queries as they are not in the hierarchy structure. However, the idea of merging each data can be applied when a pair of chosen high-frequency queries is similar. The distance function can be changed into the similarity function and the idea to merge the minimum distance pair can be changed into the most similar pair. However, this method forces us to merge until only one cluster is found. Therefore, some modification is needed as many high-frequency queries may contained in one query set.

CHAPTER III

PROPOSED METHOD

This thesis proposes methods to find a set of high-frequency queries which can be used to build similarity tables as an index structure for high-frequency-queries-based filtering. High-frequency queries can be obtained by examining the query history. Given a set of queries, it is needed to find queries which appear frequently, or are similar to many queries from this set. First, the concept of high-frequency queries is defined in section 3.1. Based on this definition, a high-frequency query is a representative of a group of similar queries. Clustering algorithms can be used to find such queries. DBSCAN is used to find high-frequency queries because it is less susceptible to noise data. From the definition, if queries are clustered densely, there are many high-frequency queries which are similar to each other. It is redundant to create two similarity tables based on two similar high-frequency queries. Two methods to choose an optimal set of high-frequency queries for high-frequency-queries-based filtering are proposed in Section 3.2. One, called DBRAN, randomly chooses one of the high-frequency queries from each cluster of high-frequency queries. The other one, called DBSM, merges similar high-frequency queries together to create a representative of these queries. The performance of DBRAN and DBSM is evaluated in Chapter 4.

3.1 Definition of High-frequency Queries

To make it possible to find useful high-frequency queries from a set of queries, a definition of high-frequency queries is formulated. First, the concept of *friends* is defined to allow the measure of frequency of a query from similar queries. Definition 1 states that a friend of a query their similarity is not lower than a specified threshold t .

Let \mathcal{Q} be a set of query containing n text queries.

Let t be a specified similarity threshold between friends in range $[0, 1]$ and min_f is a positive integer $\leq n$.

Let $f(p, q)$ be any similarity function used to calculate the similarity value between p and q .

Definition 1: A text q_1 is called a ‘friend’ of another query q_2 in iff $f(q_1, q_2) \geq t$.

Lemma 1: q_1 is a friend of q_2 iff q_2 is a friend of q_1 .

Then, a high-frequency query is defined based on the number of friends, as shown in Definition 2. The minimum number of friends and t must be chosen.

Definition 2: A text query q is called a ‘high-frequency query’ of the query set \mathcal{Q} iff q has more than min_f friends in \mathcal{Q} .

According to this definition, a high-frequency query is a query that has a sufficient number of friends. There are several ways to find high-frequency queries from a query set \mathcal{Q} . The performance of the filter method using the set found by each method is different.

3.2 Methods to Find Sets of High-frequency Queries

3.2.1 Brute Force Method

The brute force method is to find the best set of high-frequency queries by examining every combination of clusters and choosing the one that gives the best filter result. This is guaranteed to obtain the most appropriate set of high-frequency queries from the query set \mathcal{Q} as every possible set are evaluated. However, this method generates too many high-frequency queries. Thus, it requires very large

memory space to store similarity tables for high-frequency-based filtering. This makes the brute force method to find appropriate set of high-frequency queries take very long time as the time complexity is $O(n! t_n)$ if n is the number of queries in the set and t_n is the time to create the similarity tables for n records, which is $n * q \log q$ if q is the average length of texts in the dataset. Therefore, this method is impractical.

To improve the brute-force method, the number of friends of all queries can be calculated first. That is, the similarity between every pair of queries is first calculated and the number of friends is counted based on the similarity. After that, only the texts with more than min_f friends are chosen as the high-frequency queries. Therefore, the time complexity is reduced into $O(n^2 t_n)$

However, the number of high-frequency queries can be very high, and only queries with higher number of friends can be chosen. Therefore, this method is modified by choosing only max_f queries that have at least min_f friends. But this may not improve the filter power because many of these high-frequency queries are similar.

This method has two major drawbacks. One is that the query set may contain more than max_f high-frequency queries. Hence, the set obtained may not cover some queries in the set. The other disadvantage is that some query set may contain many of the similar texts. Therefore, the high-frequency queries retrieved are similar.

- The set may not cover some queries that have more than min_f friends

Obviously, if only max_f queries are chosen, some queries, with more than min_f friends, might be omitted. The groups of these queries may contain fewer queries than the one chosen but they may be parts of the main characteristics of Q . Consider figure 33, suppose figure 33 is the visualization of queries relationship. The nearer each objects located state

the more similar the texts are. There are three groups of text here represented by crosses, triangles and circles. If $max_f = 5$, only texts represented by cross object are chosen as the high-frequency queries. This is because there are more members in the group of cross' text than the others. If the similar query set is queried while only the cross text are used to construct the similarity table, texts represented by triangles and circles would not be covered. This leads to poor performance of high-frequency-queries-based filter.

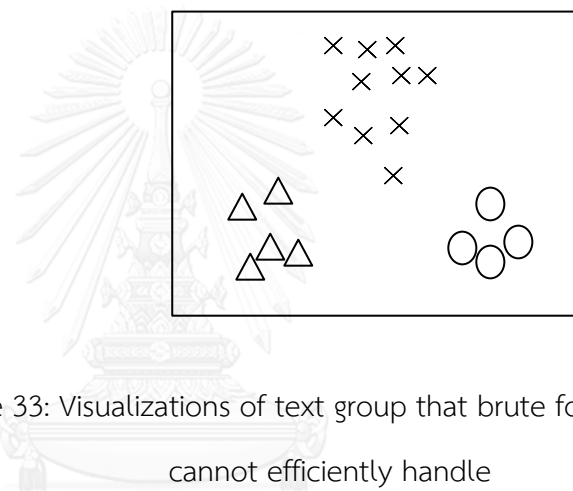


Figure 33: Visualizations of text group that brute force method cannot efficiently handle

- The high-frequency queries are similar to each other

In some situations, high-frequency queries in a query set are very similar with each other. If there are many of these high-frequency queries, the brute force method might find only them as they have many friends in the set. When these queries are used to construct the similarity table for high-frequency-queries-based filter, the similarity tables would be nearly the same for all queries. This leads to redundancy and a waste of space. If the new query set is queried, each query may suffer to choose the similarity table as the similarity value is almost equal.

Most of the query sets with high-frequency queries contain more than one unequal size groups of similar text. Therefore, many sets applied brute force method suffer from both suggested issues. To avoid such problems, we need more information about the relationship between the texts. If the details about the group of similar texts are available, we may be able to choose only some texts from each group. So, none of the smaller groups are accidentally ignored.

The task to automatically find groups of objects without any suggestions of the groups label is called cluster analysis. As mentioned in Chapter 2, there are three main types of the methods to solve this task. Two of them are applied in this section. The first one is DBSCAN.

3.2.2 DBSCAN with Random Core Points (DBRAN)

When high-frequency queries are closed together, similarity tables created from these queries can be so much alike that they are redundant. To avoid this redundancy, only some high-frequency queries must be chosen. This thesis proposes to use clustering algorithm to group queries together. DBSCAN is used to find clusters of queries, and a representative of each cluster is chosen. Based on the definition of high-frequency query, DBSCAN method can be used to find high-frequency queries with min_f is $minpts$ and t is the similarity threshold. However, the distance function used in DBSCAN is different from the similarity function. The distance between two data points is high if they are different. On the other hand, the similarity between two data points is low if they are different. According to [10], a distance function can be mapped into similarity function using three conversion functions:

linear, sigmoidal and inverted. In this section, the linear conversion function is used.

We already discussed that the DBSCAN clustering algorithm can be applied to find a set of high-frequency queries. The method to find the set of high-frequency queries by DBSCAN is illustrated in **Algorithm 1**

Algorithm 1

Input: a query set Q , a threshold t , a similarity function f , an integer min_f

Output: a set of high-frequency queries F , a set of border points B

$F \leftarrow \phi$

$B \leftarrow \phi$

for every pair q_1 and q_2 in Q

 if q_1, q_2 are friends, put them in the same cluster and put them in each other border set

$count \leftarrow$ number of border point of q_1

 if $count$ exceeds min_f

 add q_1 to F

 add border set of q_1 to B

 end if

end for

return $\langle F, B \rangle$

This method provides the information of clusters of which each core points are member. This can be used further if the set of high-frequency queries generated is very large and need to pick only the best subset to use.

We applied DBSCAN to find the set of high-frequency queries. However, sometimes core points in the same clusters are similar to each other. That leads to the redundancy in the high-frequency queries set. This can be solved by randomly choose only one text from each cluster as the

high-frequency query. The modified method is called DBRAN. The algorithm for DBRAN is described in **Algorithm 1.1**.

Algorithm 1.1

Input: a query set Q , a threshold t , a similarity function f , an integer min_f

Output: a set of high-frequency queries F , a set of border points B

```

 $F \leftarrow \phi$ 
 $B \leftarrow \phi$ 
 $P \leftarrow \phi$ 
for every pair  $q_1$  and  $q_2$  in  $Q$ 
    if  $q_1, q_2$  are friends, put them in the same cluster and put
    them in each other border set
         $count \leftarrow$  number of border point of  $q_1$ 
        if  $count$  exceeds  $min_f$ 
            add  $q_1$  to  $P$ 
            add border set of  $q_1$  to  $B$ 
        end if
    end for
for every cluster  $C$ , randomly pick one  $c \in C$  from  $P$  and add  $c$  to  $F$ 
return  $\langle F, B \rangle$ 

```

However, if the cluster is large, the random text might not cover every text in the cluster. DBSM method extends the DBSCAN by merging similar points, i.e. text data, together to reduce the redundancy and also preserve the core point coverage as much as possible. This method is described next.

3.2.3 DBSCAN with Merging Strategy (DBSM)

DBRAN deals with the problem of core point redundancy in DBSCAN by randomly choosing one core point for each cluster. However, one core point may not cover every query in the large cluster. Two dissimilar core points in the same cluster should both be chosen as high-frequency queries to capture the main characteristic of their cluster. However, sometimes two core points in the same cluster can be merged as a high-frequency query if

they are similar. Therefore, we need the criteria to determine whether a pair of core points should be picked both as the high-frequency queries or only their similar section. We propose the merging scheme to deal with both this problem and the core point redundancy problem.

For each cluster, the high-frequency queries are brought to find the pair with highest similarity. This is similar to the hierarchical agglomerative clustering. In this algorithm, the decision for merging is based on the resulting similarity table. Suppose we have two high-frequency queries d_1 and d_2 which are most similar with each other, compare with other pairs of core-points in the same clusters. The merged text query is created and it contains only the common tokens between d_1 and d_2 . Specifically, the merged query $m = d_1 \cap d_2$. Then, the number of candidates in the similarity table created with m as the high-frequency query and the number of candidates in the similarity table created with d_1 and d_2 as the high-frequency queries are compared. The candidates are obtained by doing high-frequency-queries-based filter using their friends as queries with the specified threshold t . If the latter one is larger or equal, we use this new text as the high-frequency queries instead of d_1 and d_2 . Otherwise, we do not merge them and continue trying to merge other pairs until they are all unable to merge. The remaining set is the set of high-frequency queries.

The time complexity of DBSCAN is $O(n^2)$ if n is the number of the queries in the query set. As the time complexity of agglomerative hierarchical-based clustering algorithm is $O(d^3)$ if d is the number of data in the dataset, the merging strategy based on agglomerative hierarchical clustering algorithm takes $O(km^3 t_{3m})$ if k clusters are found and m is the average number of core points per cluster. Therefore, the time complexity of DBSM is $O(n^2 + km^3 t_{3m})$.

The details of DBSM algorithm are stated in **Algorithm 2**.

Algorithm 2

Input: a query set Q , a threshold t , a similarity function f , an integer min_f , a data set D
Output: a set of high-frequency queries F
 $\langle C, B \rangle \leftarrow DBSCAN(Q, t, f, min_f)$
 $F \leftarrow \phi$
while (some pair can be merged)
 find pair c_1 and c_2 with highest similarity
 $bc_1 \leftarrow$ friends of c_1
 $bc_2 \leftarrow$ friends of c_2
 $merge \leftarrow c_1 \cap c_2$
 $bm \leftarrow bc_1 \cup bc_2$
 $Cm \leftarrow$ high-freq-based($D, f, t, merge, bm$)
 //high-freq-based(dataset, //function, threshold, a high-frequency //query, query set)
 $Cc \leftarrow$ high-freq-based(D, f, t, c_1, bc_1) + high-freq-based(D, f, t, c_2, bc_2)
 if($Cc > Cm$)
 add merge to F
 end if
 else
 set this pair similarity value = 0 //prevent further //recalculation
 end else
end while
return F

The *high-freq-based* function in **Algorithm 2** is used to find the number of candidates before and after the merging strategy is done. The example of how the function is processed is in Example 8.

Example 8: Suppose there are two texts query $q_1 = \{t_1, t_2, t_3, t_4, t_5\}$ and $q_2 = \{t_2, t_3, t_5, t_6\}$. q_1 has two friends $b_1 = \{t_1, t_2, t_3, t_4\}$ and $b_2 = \{t_1, t_2, t_3, t_4, t_6\}$ while q_2 has one friend $b_3 = \{t_1, t_2, t_3, t_4, t_6\}$. To determine whether q_1 and q_2 should be merged or not, finds $m = q_1 \cap q_2 = \{t_2, t_3, t_5\}$. Then,

compute the similarity between m and b_1, b_2 and b_3 , which is shown in figure 34 (b). The similarity table for m is created from the dataset. Suppose similarity tables for q_1, q_2 and m is shown in figure 34(a).

After that, the candidate numbers for q_1 and q_2 are calculated from ST_{q_1} and ST_{q_2} . The candidate number of q_1 using b_1 and b_2 is $1+1 = 2$ while candidate number of q_2 using b_3 is 1. Therefore, the candidate number before merge is $2+1 = 3$.

Then, the candidate number for m using b_1, b_2 and b_3 is calculated from ST_m , which is $0+4+4 = 8$. Since $8 > 4$, which means the candidate after merge is more than the candidate before merge, q_1 and q_2 will not be merge. Otherwise they are merged into m and all of their friends are added to the friend list of m .

ST_{q_1}	ST_{q_2}	ST_m
1	1	0
2	1	2
4	2	4
15	13	28
28	33	16

(a) Similarity tables of q_1, q_2 and m

	b_1	b_2	b_3
q_1	0.89	0.8	Not necessary
q_2	Not necessary		0.75
m	0.57	0.52	0.86

(b) Similarity between each texts

Figure 34: Similarity tables and similarity between texts in example 8

In this chapter, two methods are proposed to find the appropriate set of high-frequency queries as brute force methods are claimed low performance. Therefore, only DBRAN and DBSM are implemented to compare the performance in the experiment section.



CHAPTER IV

EXPERIMENT

This section describes the experiment results of proposed method to find the appropriate set of high-frequency queries. Two methods, which are DBRAN and DBSM, are implemented in Java and compile with Netbeans IDE 7.1.2 on Windows 7 professional machine with 8 GB memory.

Datasets

The datasets Enron and NYTimes from UCI machine learning databases [15] and dataset DBLP [16] are used in this experiment. The detail of each dataset is shown in figure 35.

Dataset	Number of possible words	Number of record	Average length per record
Enron	28,099	39,861	160
NYTimes	101,636	299,749	232
DBLP	467,446	1,385,952	14

Figure 35: Details of each dataset used in the experiment

Original Sets of High-frequency Queries

To evaluate the performance of the proposed methods to find the high-frequency queries from the query set, the query sets which contain high-frequency queries are needed. A query set is generated by choosing some text records from the dataset and changing some tokens in the records. The number of high-frequency queries is controlled by choosing some text records more often. Initially, from each dataset, 8, 16, 32, 64 and 128 records are randomly chosen as the original high-frequency queries which are labeled in the experiment as the 'Original' set.

Query Sets

Query sets are generated for each dataset, with different characteristics of high-frequency queries. Using the original set, high-frequency-queries based filter should give the best performance. Therefore, the results from these sets are used as the base line to compare with the results from both proposed methods.

In these chosen queries, some tokens in the queries are mutated to create the new queries with specific mutation value. Mutation can be done by inserting, deleting or substituting a token in a query. A certain percent of tokens in each query are mutated, and the mutation percentage is varied from 20% to 50%. Each query set contains mutated high-frequency queries and randomly chosen records in the dataset. The percentage of the high-frequency queries are 50, 60 and 80 percent. However, since the percentage of queries related to high-frequency queries does not affect the difference of the result between each method, only the results from 60% related sets are shown.

Half of each query set is randomly chosen as a train set that DBRAN and DBSM use to find high-frequency queries. The other half of the set is used as a test set. That is, the test set is used as query sets for high-frequency-queries based filter to measure the performance.

Performance Measure

The set of high-frequency queries obtained from DBRAN and DBSM is used to measure the performance of the two methods. The method performs well if, given the set of high-frequency queries obtained from the method, high-frequency-queries based filtering works well. Two factors – coverage and the number of candidates - indicate that high-frequency-queries based filtering works well. The coverage of a set of high-frequency queries is the number of queries in a query set that similar to at

least one high-frequency query. These queries are called *in-coverage queries* while the rest are called *out-of-coverage queries*. Large coverage indicates that the similarity tables can be used for many queries. When a query is in-coverage, a similarity table is used and the candidates are obtained from the table. These candidates are called *in-coverage candidates*. A smaller number of in-coverage candidates indicates that the filter works well. On the other hand, when a query is out-of-coverage, high-frequency-queries based filter does not use similarity tables and switches to AdaptSearch. In this case, the candidates obtained from similarity tables, called *out-of-coverage candidates*, are too numerous to be of use. It is preferable for a set of high-frequency queries to have large coverage large and a small number of in-coverage candidates because this makes the number of candidates for each query very small. Thus, the number of in-coverage candidates per in-coverage query is also an important indicator.

4.1 Coverage Percentage

The coverage percentage is the percent of the in-coverage queries in a query set. High coverage percentage indicates that many queries are similar to at least one of the chosen high-frequency queries. In this case, a similarity table is used for filtering, and the number of candidates should be small. Thus, it is desirable that a set of high-frequency queries gives high coverage percentage.

To show that the sets of high-frequency queries obtained from the improved brute force method gives low coverage percentage, it is compared with the results from DBRAN and DBSM, when applied on DBLP. The result, shown in figure 36, indicates that the high-frequency queries found by the brute force method cover fewer queries than those from both proposed method.

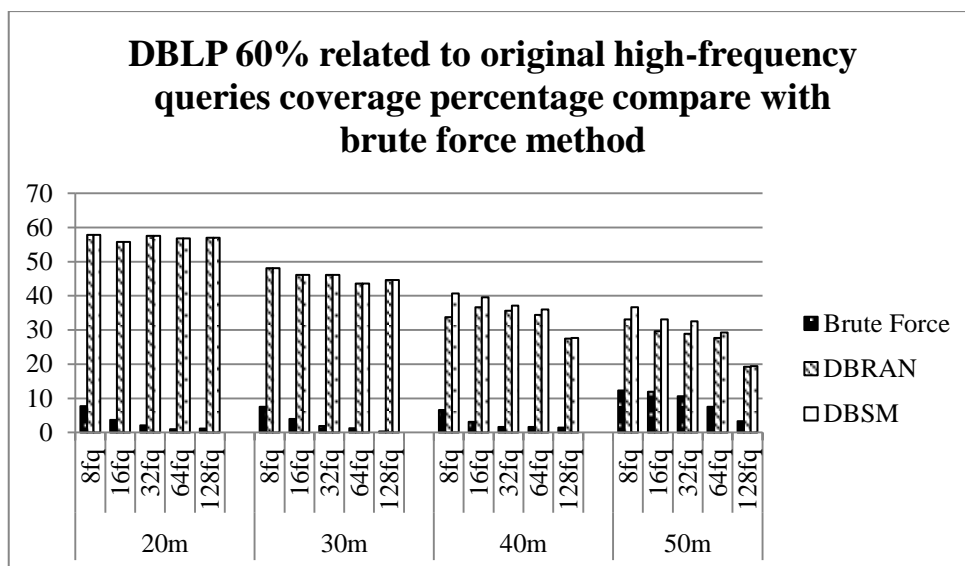


Figure 36: Coverage percentage of DBLP with 60% related to original high-frequency queries compare with brute force method

Figure 37-39 shows the coverage percentage from 3 datasets when 60% of the queries are related to the original high-frequency queries. The charts show that, when the mutation is low, i.e. at 20%, the coverage percentages of high-frequency queries obtained from DBRAN, DBSM and original sets are nearly the same. This means sets of high-frequency queries retrieved from these two methods similar with the original sets. With at least 40% mutation, the coverage percentages from DBSM sets are better than DBRAN but lower than original sets. For the original set of high-frequency queries, it remains the same when the queries are mutated at lower level. On the other hand, if the queries are mutated more than 40%, the coverage percentage decreases when the mutation level increases.

Another point to consider is the effects of the number of original high-frequency queries. If queries are mutated 20%-30%, numbers of high-frequency queries do not affect the coverage percentage. On the other hand, coverage percentage of the DBSM and DBRAN results moderately decrease if the train sets

contain more high-frequency queries. However, the original sets of high-frequency queries are not affected by this parameter.

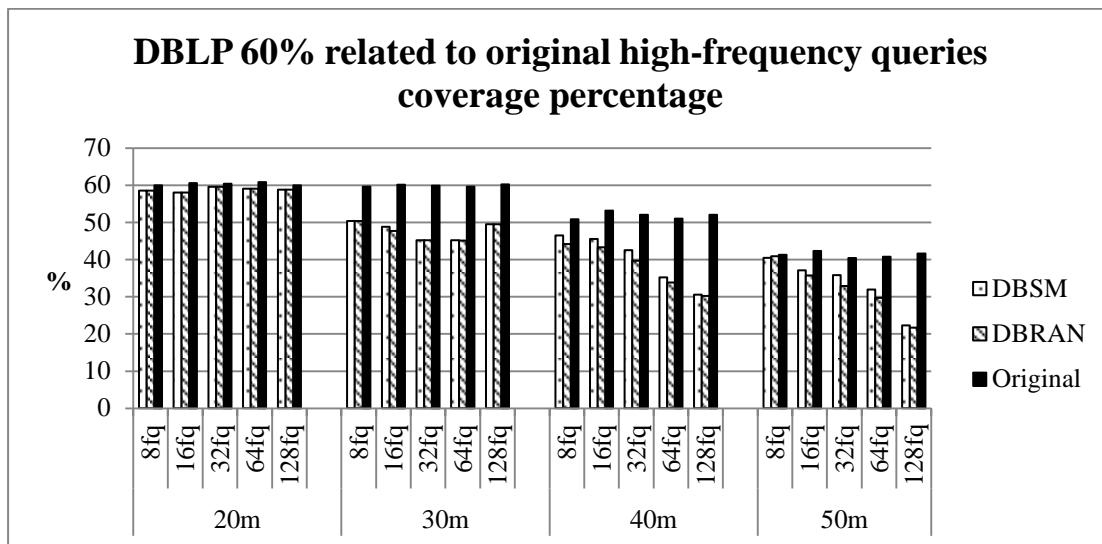


Figure 37: Coverage percentage of DBLP with 60% related to original high-frequency queries

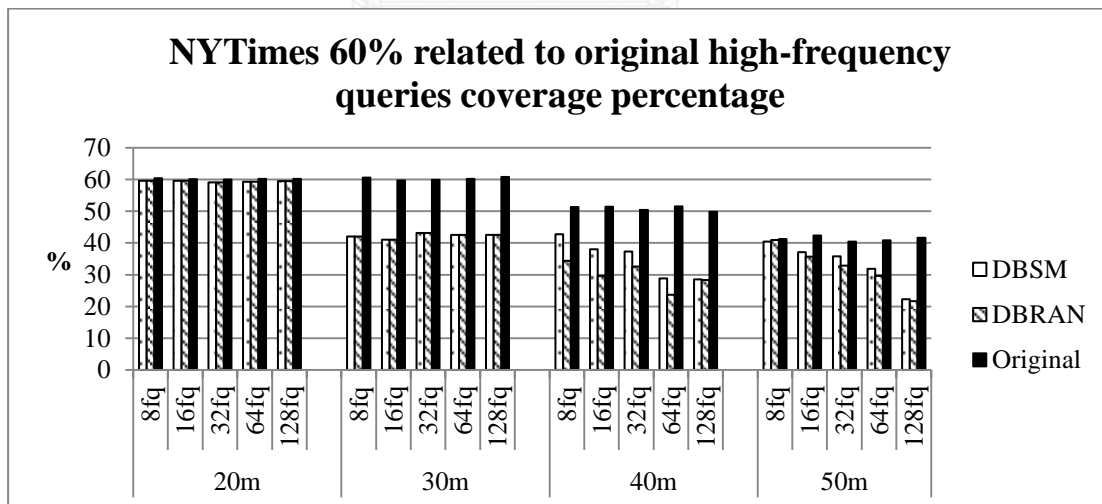


Figure 38: Coverage percentage of NYTimes with 60% related to original high-frequency queries

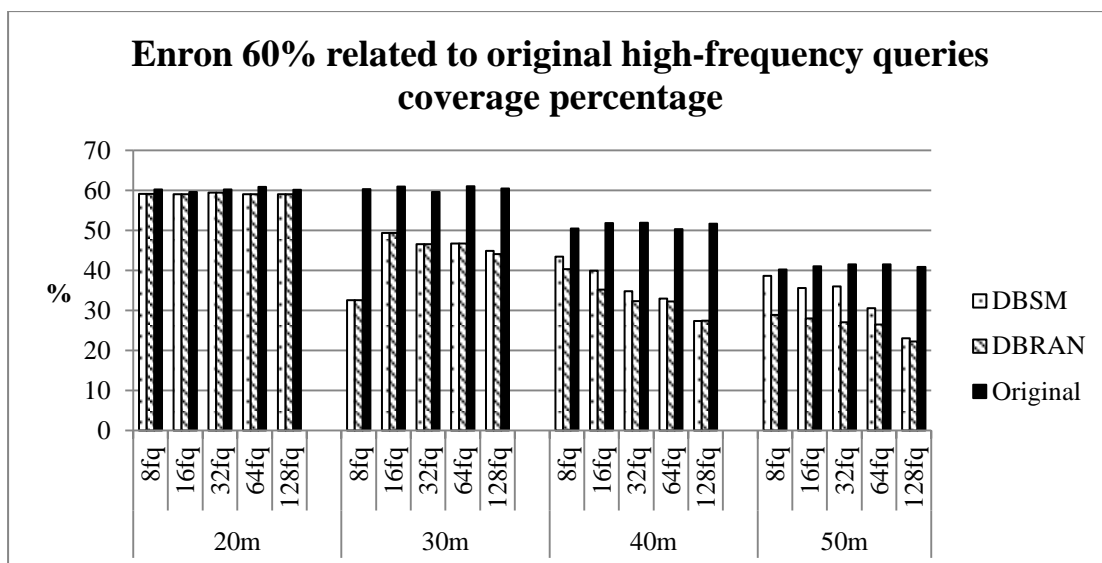


Figure 39: Coverage percentage of Enron with 60% related to original high-frequency queries

The higher number of in-coverage queries lead to higher in-coverage candidates which is shown next section.

4.2 Numbers of In-coverage Candidates

If a query is in-coverage, its candidates are retrieved from the similarity table of the closest high-frequency query. The number of in-covered candidates depends on the similarity threshold, which is 0.5 in this experiment, and the similarity between each query and the chosen high-frequency query. Therefore, the number of in-coverage candidates indicates how much verification is required for similarity join. Figure 40-42 shows the in-coverage candidate number of every datasets when 60% of train sets are related to the original high-frequency queries.

For every datasets, when the queries are 20% mutated, the number of in-coverage candidates are low. The number of in-coverage candidates from DBSM and DBRAN high-frequency queries is highest when the queries are 30% mutated and then decrease when queries are mutated more. In contrast, in-coverage candidates

of original high-frequency query set dramatically increase from 20% mutation level and reach the highest value at 40% query mutation.

In contrast, in-coverage queries of the original sets generate very few candidates when the queries are mutated 20%-30%. However, the in-coverage candidate numbers dramatically increase to almost the same with DBSM and DBRAN when the queries are 40% mutated. With 50% query mutation level, the in-coverage candidate number decreases as well as the result from other methods. This means out-of-coverage candidates increase as the queries are more mutated.

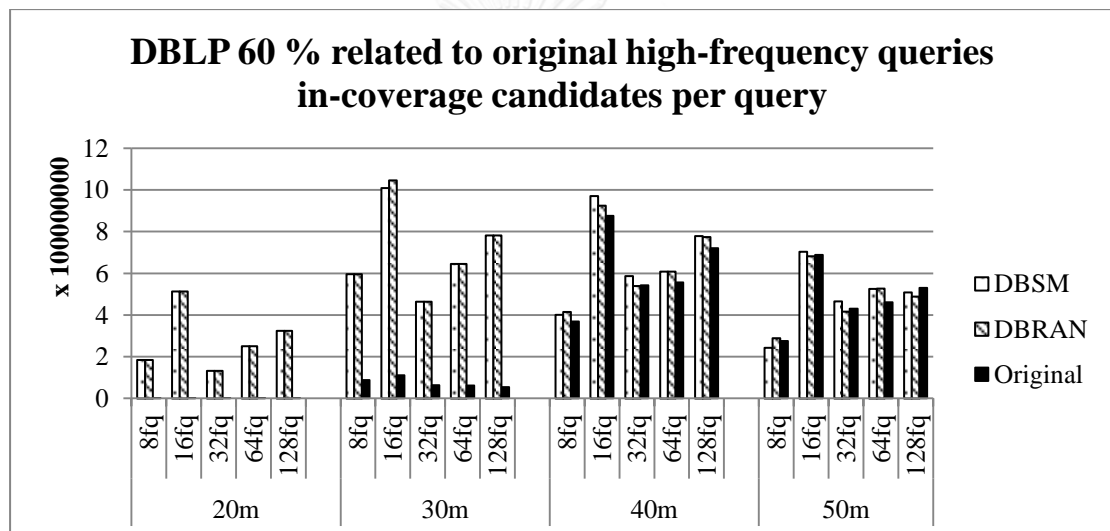


Figure 40: In-coverage candidates of DBLP with 60% related to original high-frequency queries

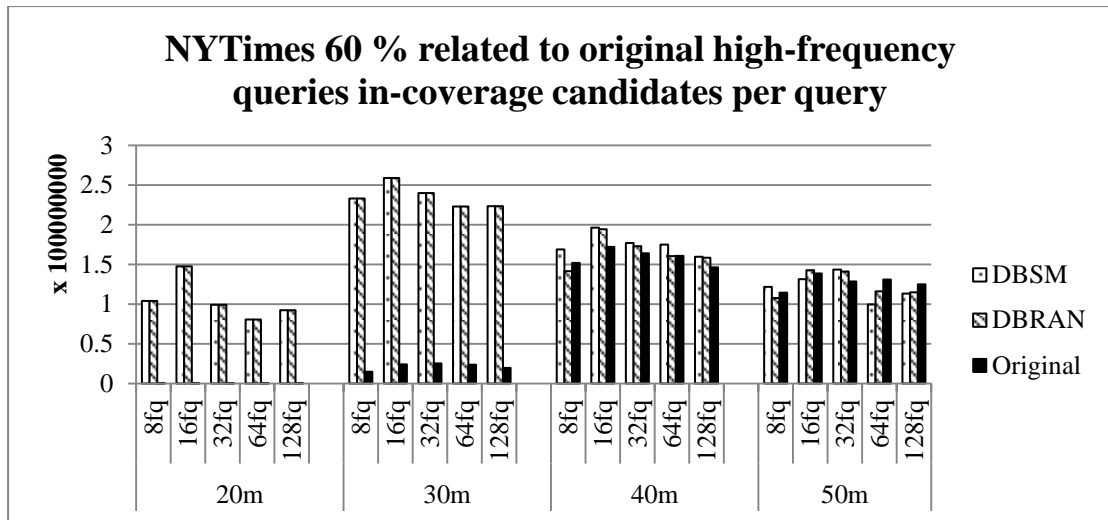


Figure 41: In-coverage candidates of NYTimes with 60% related to original high-frequency queries

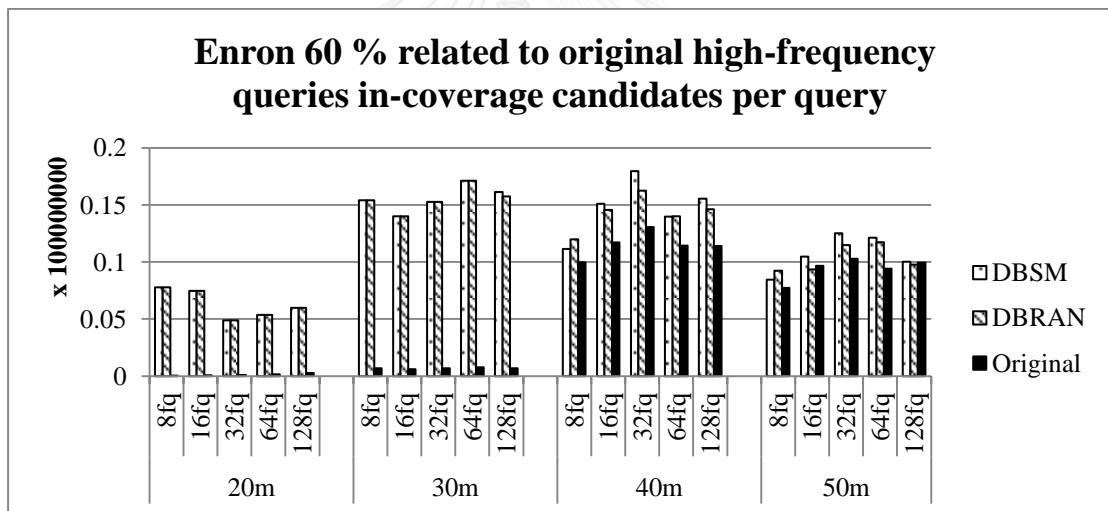


Figure 42: In-coverage candidates of Enron with 60% related to original high-frequency queries

4.3 Average Number of Candidates per In-coverage Query

The total number of in-coverage candidates increase with more in-coverage queries. Therefore, candidate numbers alone are not enough to judge the performance of the retrieved high-frequency queries. The average number of candidates generated by each in-covered query must also be considered. This is

computed from the number of all in-coverage candidates divides by the number of in-coverage queries. As the value is different according to the total number of records in each dataset, it is shown as the percentage compared with overall dataset instead.

Figure 43-45 shows the mentioned percentage of each dataset when 60% of the test sets related to the original high-frequency queries with various mutation levels. For every dataset, if the query is 20% mutated, the in-coverage queries using original high-frequency queries generate significantly low amount (less than 0.01%) of in-coverage candidates from the whole dataset while both DBSM and DBRAN values are bounced from 2%-6%. When the query set are more mutated, DBSM and DBRAN high-frequency queries generated in-coverage candidates per query at nearly the same level. In contrast, with the highly mutated queries, says more than 40%, each of the in-coverage queries using the original high-frequency query sets generate dramatically higher candidates.

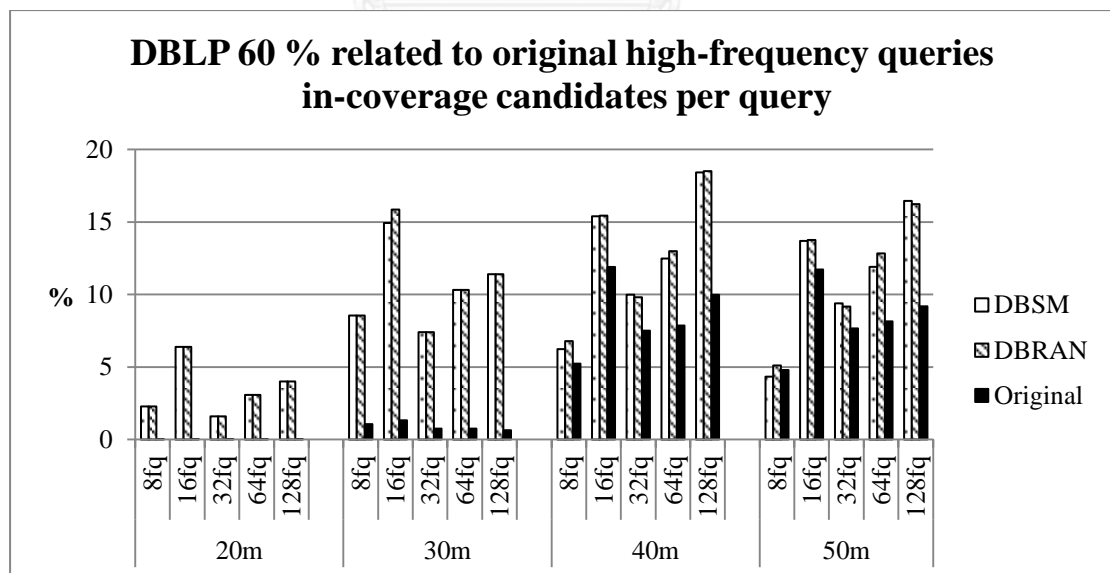


Figure 43: In-coverage candidates per query percentage of DBLP with 60% related to original high-frequency queries

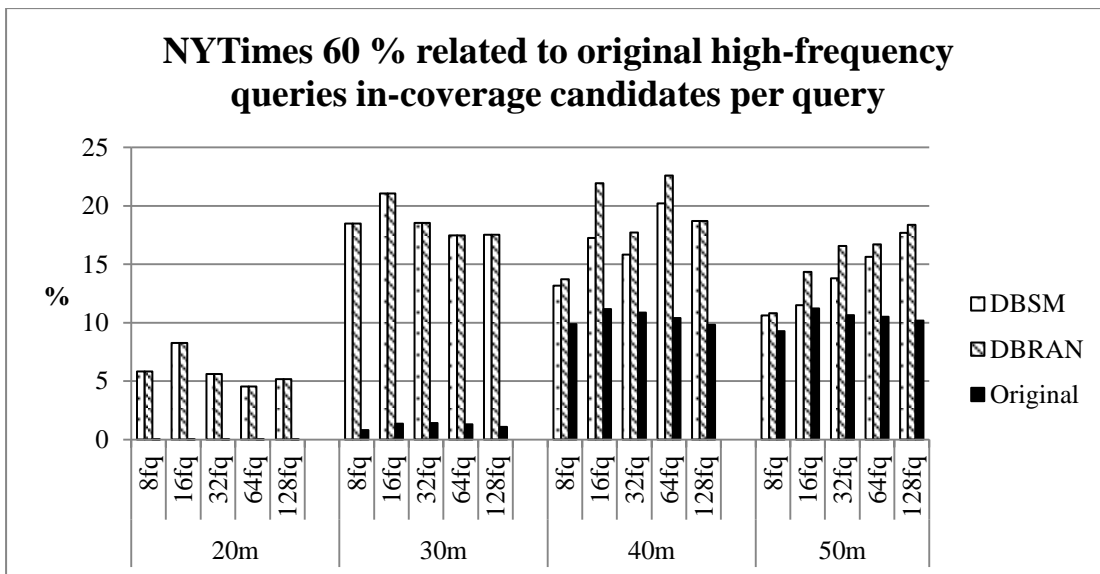


Figure 44: In-coverage candidates per query percentage of NYTimes with 60% related to original high-frequency queries

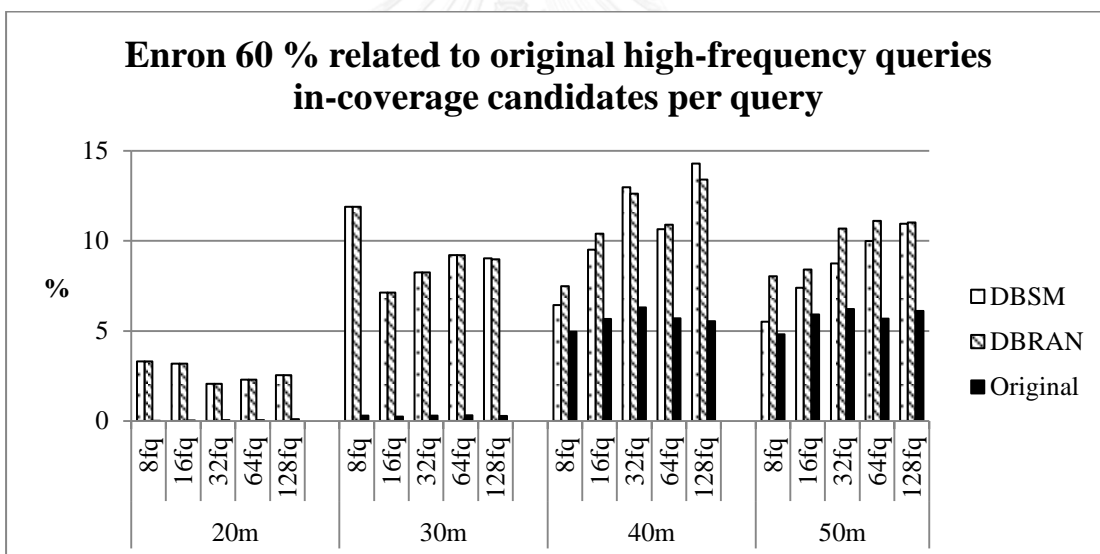


Figure 45: In-coverage candidates per query percentage of Enron with 60% related to original high-frequency queries

The results from these three measurements lead to two important conclusions. First, if the queries are mutated at low level, high-frequency-queries-based filter using the high-frequency queries found from DBRAN and DBSM give the similar results. Query set with low mutation contains many similar queries which lead to denser cluster. Therefore, the merging strategy is not necessary as the random

strategy still performs well. On the other hand, if the queries are highly mutated, DBRAN and DBSM give slightly different sets of high-frequency queries which lead to moderately different in coverage percentage, candidate numbers and the percentage of candidate numbers per query in the dataset. The sets obtained from DBSM provide better results in most of the experiment. This means that the merging strategy is necessary for the highly mutated query set.



CHAPTER V

CONCLUSION

This thesis proposes a method to find a set of high-frequency queries from a query set. The set of high-frequency queries is used to create a similarity table for high-frequency-queries-based filter in filter-and-verify framework for similarity join. DBSCAN clustering algorithm is applied to find the clusters of queries. DBRAN, which is DBSCAN with random core points, finds high-frequency queries and removes redundant core points by randomly selecting one core point from each cluster. However, one core point may not cover every query in a cluster. DBSM, which is DBSCAN with merging strategy, removes redundant core points and also preserves the coverage of each core points. This method merges two core points if they are similar. Then, the remaining core points are used as high-frequency queries. Experiment results show that DBSM and DBRAN are nearly the same when the high-frequency queries are similar, or the clusters are compact. On the other hand, if the high-frequency queries are highly varied, DBSM outperforms DBRAN as the resulting sets of high-frequency queries provide better performance for high-frequency-queries-based filter.

Although DBSM found the sets that cover more queries in the test sets than DBRAN, it takes much longer to compute if there are many core points in each cluster. Therefore, the strategy to determine whether the merging strategy is necessary for the set of core points should be studied further.

REFERENCES

- [1] M. Hadjieleftheriou and D. Srivastava, "Approximate String Processing." in Foundations and Trends in Databases, pp. 267-402, 2011.
- [2] S. Chaudhuri, V. Ganti, and R. Kaushik, "A primitive operator for similarity joins in data cleaning." in *Proceedings of International Conference on Data Engineering (ICDE)*, pp. 5-16, 2006.
- [3] J. Wang, G. Li and J. Feng, "Can we beat the prefix filtering?: An adaptive framework for similarity join and search." in *Proceedings of ACM Management of Data (SIGMOD)*, pp. 85-96, 2012.
- [4] K. Kunanusont and J. Chongstitvatana, "An Index Structure for Similarity Join Based on High-frequency queries" in *Proceedings of International Computer Science and Engineering Conference (ICSEC)*, pp. 415 – 420, 2014
- [5] J. Han and M. Kamber, *Data Mining: Concepts and Techniques* 2nd edition, Morgan Kaufman Publishers Inc., CA
- [6] M. Ester, H. Kriegel, J. Sander and X. Xu, "A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise" in *Proceedings of the second International Conference on Knowledge Discovery and Data Mining (KDD-96)*, pp. 226-231, 1996.
- [7] C. Xiao, W. Wang, X. Lin, J. Xu Yu and G. Wang, "Efficient Similarity Joins for Near Duplicate Detection." In *Proceedings of international conference on World Wide Web (WWW' 08)*, pp. 131-140, 2011.
- [8] "April 2015 Nepal Earthquake"
http://earthquake.usgs.gov/earthquakes/eventpage/us20002926#general_summary
- [9] "Google trends explorers" <http://www.google.com/trends/explore#cmpt=q>
- [10] M. Setten, M. Veenstra, A. Nijholt and B. Dijk, "Cased-Based Reasoning as a Prediction Strategy for Hybrid Recommender Systems" in *Proceeding of Atlantic Web Intelligence Conference (AWIC)*, pp. 13-22, 2004

- [11] H. Gupta and R. Srivastava. “k-means Based Document Clustering with Automatic “k” selection and Cluster Refinement” in *Proceedings of International Journal of Computer Science and Mobile Applications (IJCSMA)*, pp. 7-13, 2014.
- [12] P. Bradley and U. Fayyad, “Refining Initial Points for K-Means Clustering” in *Proceedings of the 15th International Conference on Machine Learning (ICML98)*, pp. 91-99, 1998
- [13] A. Barakbah and Y. Kiyoki, “A Pillar Algorithm for K-means Optimization by Distance Maximization for Initial Centroid Designation” in *Computational Intelligence and Data Mining (CIDM)*, pp. 61-68, 2009
- [14] C. Manning, P. Raghaven and H. Schutze, *Introduction to Information Retrieval*, Cambridge University Press
- [15] “UCI Machine Learning Repository” <https://archive.ics.uci.edu/ml/datasets.html>
Obtained on 8th January 2015
- [16] *The DBLP bibliography of published researchers in computer science* obtained on 25th November 2013 from
<http://www.cs.berkeley.edu/~jnwang/codes/adapt.tar.gz>

VITA

Miss Kamolwan Kunanusont was born on July 13th, 1992 in Bangkok, Thailand. In 2014, she received her first class honour Bachelor degree in Computer Science from Chulalongkorn University. After graduation, she pursued her graduate study for Master's Degree in Computer Science and Information Technology at Department of Mathematics and Computer Science, Faculty of Science, Chulalongkorn University in 2014. Her paper "Finding a Set of High-frequency Queries for High-frequency-query-based Filter for Similarity Join" is accepted for oral presentation on June 26, 2015 at 2015 12th International Conference on Electrical Engineering/Electronics, Computer, Telecommunications and Information Technology (ECTI-CON), Hua Hin, Thailand.

