19th International Computer Science and Engineering Conference (ICSEC)
Chiang Mai, Thailand, 23-26 November, 2015

# Refining High-frequency-queries-based Filter for Similarity Join

Jaruloj Chongstitvatana
Department of Mathematics and Computer Science
Faculty of Science, Chulalongkorn University
Bangkok 10330, Thailand
jaruloj.c@chula.ac.th

Natthee Thitinanrungkit
Department of Mathematics and Computer Science
Faculty of Science, Chulalongkorn University
Bangkok 10330, Thailand
n.thitinanrungkit@gmail.com

***Abstract*—Similarity join and similarity search are important for text databases and data cleaning. Filter-and-verification are applied to reduce the processing time for similarity join and similarity search. High-frequency-queries-based filter partitions a dataset according to the similarity between a data record and a chosen high-frequency-query, and these partitions are stored in a similarity table. In the filter process, data in some rows of a similarity table are selected as candidates. Many high-frequency queries can be used to improve the pruning power. However, the time to choose an appropriate high-frequency query – i.e. to choose an appropriate similarity table – increases with the number of high-frequency queries.**

**This paper proposes a refinement of high-frequency-queries-based filter to reduce the filter time and the number of candidates. To reduce the filter time, inverted lists of high-frequency queries are used to speed up the token counting, which reduces the time for choosing an appropriate similarity table. Binary search in each rows of a similarity table is applied to further eliminate non-candidates. It is shown from the experiments that the refined filter method takes less time and gives better pruning power than the original method.**

***Keywords—similarity join; filter-and-verfication approach; high-frequency queries;***

## I. INTRODUCTION

In text databases, text can be represented by a sequence of characters or a sequence of tokens, and each token represents a set of words that can be considered the same [1]. In many applications, text is represented by a set of tokens. The similarity between a pair of text is measured from similarity functions [2], e.g. overlap similarity, Jaccard similarity, cosine similarity, etc. These functions are based on the number of common tokens in two text data and the number of tokens in each text data.

Similarity join is an operation which finds pairs of similar text from two relations, and it is used in text databases, data cleansing, data integration, etc. [3] However, calculating the similarity for all possible pairs of text is costly, especially when the datasets are large and text data are long.

The filter-and-verification framework [3] is used to filter out some unlikely pairs of text data without calculating their similarity. For all remaining pairs of text data, their similarity is verified by calculating the similarity function. Some filtering techniques, such as prefix filtering [3, 4], suffix filtering [5], positional filtering [5], examine only some part of text data in order to spend the least amount of time while filtering out as many non-related text as possible.

Another filtering technique, called high-frequency-queries-based filtering [6], partitions a dataset based on the similarity between each text data and a chosen text, and stores in a table called similarity table. This chosen text is called a high-frequency query because it is a text data which appears in high frequency, or is similar to many queries. The dataset is filtered for each query based on the similarity between the query and the chosen high-frequency query.

Normally, more than one high-frequency queries are chosen in order to improve the pruning power, and the filter cost grows linearly with the number of chosen high-frequency queries. The cost of calculating the similarity between the query and a high-frequency query is high if the query or the high-frequency query are long. This increases the filter time. Furthermore, the filter power depends partly on the grain of similarity tables, but it is inefficient to use fine-grain tables. This work proposes a refinement of high-frequency-queries-based filtering to reduce the filter time and increase the pruning power. A data structure is created to speed up the process of choosing the high-frequency query. The similarity table is modified to support finer grain of data partition and improve the filter power.

This paper is first described filter methods for similarity join, especially high-frequency-queries-based filtering in Section II. Then, the refinement of high-frequency-queries-based filtering is proposed in Section III. Section IV describes the experiments performed to compare the proposed refinement to existing methods, together with the result. The conclusions are discussed in Section V.

## II. RELATED WORKS

The brute-force approach for similarity join is prohibitively expensive because it is expensive to compute the similarity between two long texts, and computing the similarity between all pairs of texts is even more costly. Filter-and-verification framework reduces the cost of computing the similarity by first filtering out non-candidates, which are text that cannot possibly

978-1-4673-7825-3/15/$31.00 ©2015 IEEE

be answers of the query. Then, the similarity of the remaining candidates is computed in the verification step. Only the candidate whose similarity exceeds the given threshold is returned as an answer. Many approaches are used in the filter step. Many filter methods, such as prefix filtering, suffix filtering and positional filtering, examine only some part of a text and determine if the text is a candidate or not. Another approach, i.e. high-frequency-queries-based filter, divides text data into partitions, and chooses only some partitions as candidates according to the query.

### *A. Prefix Filtering*

Prefix filtering examines only the prefix of each text pair and finds out whether the number of common tokens between two texts exceeds the specific threshold before determines to keep or prune. It is possible to determine if a text record $r$ contains less than $o$ tokens in common with a text query $q$ by examining the first $|r|-o+1$ tokens of $r$. Thus, a prefix of that given length is examined. If it is found that the text record cannot possibly contains more than the required common tokens, it is filtered out. Otherwise, it remains a candidate for the query, and is examined further in the verification step.

In [3], a fixed prefix length is used for all data. The pruning power can be improved with longer prefix at the cost of computing time. However, there is no optimal prefix length for every string. [4] proposes an adaptive method, called AdaptSearch and AdaptJoin, for determining an optimal prefix length, based on the estimated cost vs. the estimated pruning power of increasing prefix length. If the increased pruning power outweighs the increased cost, then the prefix length used in filtering is increased.

### *B. Positional Filtering*

Positional filtering considers the position of each token in a pair of text to estimate the highest similarity between the pair. Positional filtering is used together with other filtering methods in Ppjoin+ [5].

### *C. Suffix Filtering*

Suffix filtering uses the suffix of a text to filter. The suffix of a text 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. The more the suffix filtering is recursively applied, the more candidates can be pruned. Therefore, trade-off between the cost of filter and verify must be considered for the overall performance.

Ppjoin+ uses prefix filtering, positional filtering and suffix filtering. 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 specify the number of times that suffix filtering is recursively called, and the computation cost and the pruning power can be traded off.

### *D. High-frequency-queries-based Filtering*

This method is based on the assumption that there are some queries which are often asked, as shown in Google trends explorer [7]. High-frequency-queries-based filtering groups similar text records together according to their similarity with respect to a chosen text, which is a query that is frequently asked. This query is called a high-frequency query.

An index structure, called a similarity table, is created to store pointers to all texts in the dataset. Text data are organized in the similarity table according to the similarity between the high-frequency query and the text data. Given a high-frequency query $F$ of a dataset $D$ and the similarity table $T$ created from $F$ with $s$ rows, the row $i$ of $T$ stores the pointers to the data records whose similar value, compared to $F$, is between $i/s$ and $(i+1)/s$. That is, $T[i] = \{p \mid p$ is the pointer to $r \in D$, $i/s < sim(r, F) \le (i+1)/s\}$. For example, given a similarity table with 5 rows, the row $i$ contains data records whose similarity with respect to the high-frequency query is between $i/5$ and $(i+1)/5$. To improve the pruning power, many high-frequency queries can be used. For each high-frequency query $hf$, a similarity table is created, and the dataset in each similarity table is partitioned according to the similarity between the data record and $hf$. For a query, the high-frequency query which is most similar to the query is chosen, and the corresponding similarity table is used for filtering.

The filter algorithm is described in Figure 1. The function $index(s)$ returns the row number of the similarity table containing data records whose similarity compared to the high-frequency query is $s$. For example, given a similarity table with 5 rows, $index(0.86)$ returns the row number 5.

Given a query $Q$ with a threshold $t$, high-frequency queries $f_i$ and a similarity table $T_i$ which stores partitions of data according to the similarity between each data record and $f_i$.

1. [Find the best similarity table for filtering]
   For each high-frequency query $f_i$ , find $sim(f_i, Q)$.
   Choose a high-frequency query $hf$ such that $sim(hf, Q) \ge sim(f_i, Q)$ for all high-frequency queries $f_i$.
   $ST$ := the similarity table created from $hf$.
2. [Use similarity table for filtering]
   If $sim(hf, Q) \ge 0.5$,
   $up := index(sim(f_h, Q) + t + 1)$
   $low := index(sim(f_h, Q) + t - 1)$
   $candidates = \{\ \}$
   for $r := low$ to $up$
   $candidates := candidates \cup ST[r]$.
3. [Use adaptSearch for filtering]
   If $sim(hf, Q)<0.5$, $candidates := adaptSearch(Q)$.

Fig. 1. Algorithm for high-frequency-query-based filtering.

A problem with high-frequency-queries-based filtering is how to choose high-frequency queries so that filtering performs well for majority of the queries. [8] proposes DBSCAN, combined with cluster merging, to choose a set of high-frequency queries.

Another problem for this filter method is the cost of finding the best similarity table, shown in Step 1 of the algorithm shown in Figure 1. Although the increase in the number of

high-frequency queries can improve the pruning power, calculating the similarity between a query and all high-frequency queries is costly, especially when queries are long. This makes it impractical to use many high-frequency queries.

Another performance issue is resulted from the fixed number of rows in the similarity table. In Step 2 of the algorithm in Figure 1, after *up*, which is the top row in the range, and *low*, which is the bottom row in the range, are calculated, all data records in the rows from *low* to *up* are considered as candidates. Given a query $Q$ with the threshold $t$, the similarity between a candidate for $Q$ and the high-frequency query must be between $sim(f_h, Q) + t + 1$ and $sim(f_h, Q) + t - 1$. In Step 2, the bounds of the required similarity value is calculated and mapped into the corresponding rows in the similarity table. For example, consider a similarity table with 5 rows. If the required range of the similarity value is between 0.60 and 0.86, the candidates are in rows 3 and 4. However, the range of the similarity of data records in row 4 is between 0.8 and 1.0, and it can contain data records whose similarity which is not in the required range. These data records must be removed in the verification step and the performance of this filter method is compromised.

Next, the modification in high-frequency-queries-based filter to address these two problems is described next.

## III. Improving High-frequency-query-based Filter

This paper addresses two problems of high-frequency-queries-based filtering. The first issue is to reduce the time in choosing the similarity table for filtering. The existing algorithm uses the brute-force approach which takes a long time to compute the similarity between many pairs of long queries, and it takes too long a time when more high-frequency queries are added. The second refinement is to reduce the number of candidates obtained from the similarity tables. The finer grain the similarity table is divided, the more precise the candidates can be selected.

The modified algorithm is shown in Figure 2, and the refinement in the algorithm is described next.

### A. Faster Selection of Similarity Tables

To reduce the time for selecting the similarity table, we choose the high-frequency query with the most common tokens with the query. Inverted lists of tokens are created for all high-frequency queries to help in finding common tokens. The inverted list of a token *tkn* is a list of all high-frequency queries containing the token *tkn*. For a high-frequency query $f_j$, the number of tokens which are also in the query is stored in $HFCnt[j]$ as shown in Step 1 of the algorithm shown in Figure 2. Then, the high-frequency query with the most tokens in common with the query is chosen.

With this modification, there is no need to compute the similarity between the query and all high-frequency queries. It makes the filtering scale better for a large number of high-frequency queries.

Given a query $Q$ with a threshold $t$, high-frequency queries $f_i$, inverted lists of tokens in all high-frequency queries, and a similarity table $T_i$ which stores partitions of data according to the similarity between each data record and $f_i$.

1. [Find the best similarity table for filtering]
   for all high-frequency query $f_i$, $HFCounter[i] := 0$.
   for each token *tkn* in the query $Q$,
   for each query $f_j$ in the inverted list of token *tkn*
   $HFCnt[j] := HFCnt[j] + 1$
   $hf :=$ the high-frequency query $f_k$ such that, for all high-frequency queries $f_i$, $HFCnt[k] \geq HFCnt[i]$.
   $ST :=$ the similarity table based on $hf$.
2. [Use similarity table, with modification, for filtering]
   If $sim(f_h, Q) \geq 0.5$,
   $up := index(sim(f_h, Q) + t + 1)$
   $low := index(sim(f_h, Q) + t - 1)$
   $candidates := partition(T_h[low]) \cup partition(T_h[up])$
   for $r := low+1$ to $up-1$
   $candidates := candidates \cup T_h[r]$.
3. [Use adaptSearch for filtering]
   If $sim(f_h, Q)<0.5$, $candidates := adaptSearch(Q)$.

Fig. 2. Modified algorithm for high-frequency-query-based filtering.

### B. Reducing Candidate Sets

Another problem with the similarity table is that it partitions data records roughly by their similarity with respect to the chosen high-frequency query. Then, some data records in the top and the bottom rows, given the bound $[sim(f_h, Q)+t-1, sim(f_h, Q)+t+1]$, may be out of the bound.

We propose a modification of the similarity table by storing data records in each row in the order of the similarity. Then, data records which are out of the required bound can be eliminated with binary search, as shown in the function *partition* in Step 2 of the algorithm shown in Figure 2. That is, the similarity value of the record at the middle of the top/bottom row is calculated and it can be determined if the upper/lower half of the row can be eliminated. This process is repeated until all out-of-bound data records can be eliminated.

Experiments are done to evaluate the performance of the proposed refinement, and they are presented in the next section.

## IV. Pereformance Evaluation

In this section, experiments which are performed to evaluate the performance of the proposed modification. Two datasets – DBLP [9], which contains 1,385,925 records and 467,454 tokens, and NYTimes [10], which contains 299,752 records and 101,636 tokens – are used in the experiments. Different query sets are randomly generated from the datasets with varying percentage of high-frequency queries. Each query set contains 10,000 queries. In a generated query set, a number of high-frequency queries are created from records in the dataset by changing some tokens in the record. Each high-frequency queries is 90% similar to a record in the dataset.

First, the proposed method is compared to the original high-frequency-queries-based filter [6]. The filter time and the average filter percentage, which is the percentage of the records obtained from the filtering over the whole data records, are used to compare the performance of the two methods. Next, the proposed method is compared to AdaptSearch [4], and the query time is used as a measurement.

### *A. Comparison to the Original High-frequency-queries-based Filter*

In the experiments, the numbers of high-frequency queries and the percentage of high-frequency queries in the query sets are varied. The original and the modified high-frequency-queries-based filters are applied on DBLP and NYTimes datasets to measure the filter time and the filter percentage in the query set.

Figure 3 shows the filter time for both methods with varying number of high-frequency queries, and Figure 4 shows the filter time for both methods with varying percentage of high-frequency queries in the query set. It is shown that the proposed refinement improves the filter time. From Figure 3, the filter time increases slightly when the number of high-frequency queries increases. From Figure 4, the filter time increases when the percentage of high-frequency queries decreases.

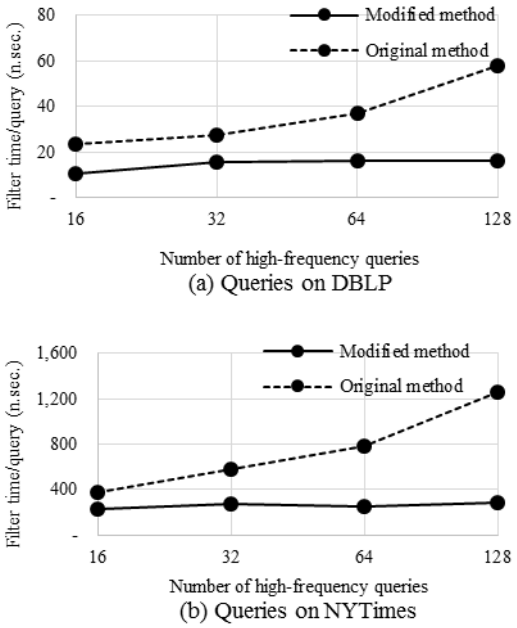


Fig. 3. Filter time for queries with varying number of high-frequency queries.

The average filter percentage is measured for similarity tables with different numbers of high-frequency queries, and for query sets with different percentage of high-frequency queries for queries. It is found that the proposed refinement does not change the filter percentage much. This can be resulted from the characteristics of data distribution in the datasets. Furthermore, the filter percentage of the proposed modification does not depends on the number of high-frequency queries used in filtering and the percentage of high-frequency queries in the query sets. However, the number of high-frequency queries effects the total query time as shown next.

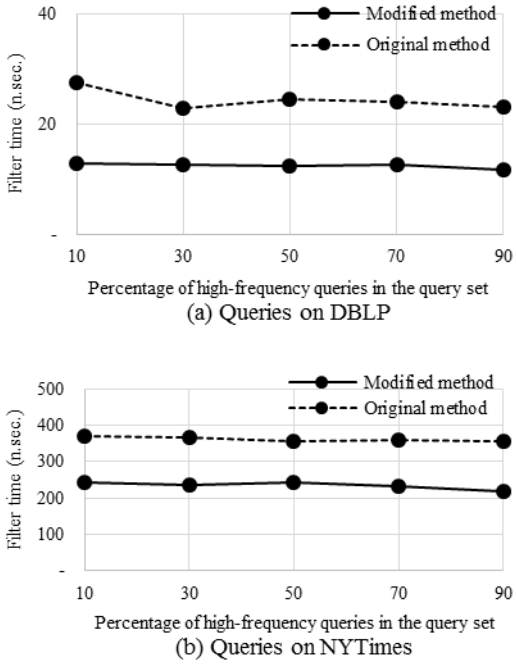


Fig. 4. Filter time for query sets containing varying percentage of high-frequency queries.

### *B. Comparison to AdaptSearch*

Only DBLP is used in the comparison to AdaptSearch because AdaptSearch's code [11] does not accommodate large datasets. The query time, which is the filter time together with the verification time, is measured for the two methods, with varying number of high-frequency queries and varying percentage of high-frequency queries.

The query time for the proposed method, using different number of high-frequency queries, for a query set with 50% high-frequency queries which are 90% similar to a record in DBLP is shown, in Figure 5. The query time on queries with varying percentage of high-frequency queries is shown in Figure 6.

From the experiment, it is shown that the proposed modification outperforms AdaptSearch especially when the number of high-frequency queries is large, and when higher percentage of queries are similar to one of the high-frequency queries.

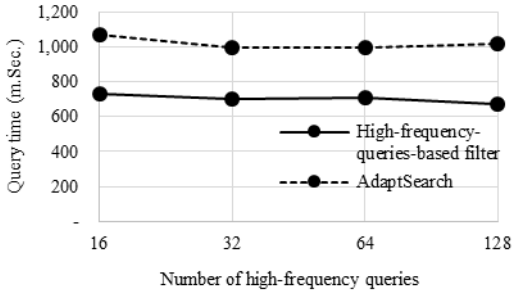


Fig. 5. Query time with varying number of high-frequency queries.

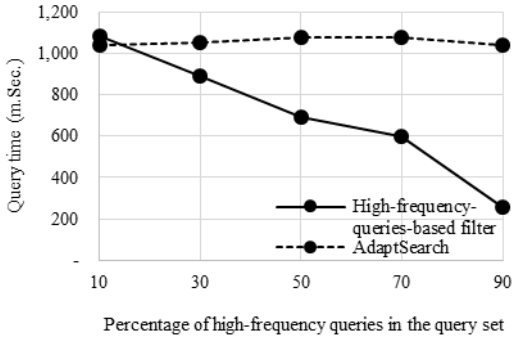


Fig. 6. Query time for query set containing varying percentage of high-frequency queries.

## V. CONCLUSION

This work proposes a refinement for high-frequency-queries-based filtering. First, we make the selection of similarity tables faster. Inverted lists of high-frequency queries are used to speed up counting the number of tokens in high-frequency-queries which are also in the query. Second, the sets of data obtained from the upper-bound row and the lower-bound row of the similarity tables are further filtered by binary search.

It is shown in the experiments that the modified filter method is faster than the original one, and the filter percentage of the modified filter method is also better than the original filter method. Moreover, the filter time increases slowly with the increase in the number of similarity tables. Finally, when this modification of high-frequency-queries-based filtering is used in similarity search, this method is faster than AdaptSearch.

## REFERENCES

[1] N. Augsten and M. H. Bohlen, "Similarity Joins in Relational Database Systems", Synthesis Lectures on Data Management, vol. 5, no. 5, p. 1-124, 2013.

[2] M. Hadjieleftheriou and D. Srivastava, "Approximate String Processing." in Foundations and Trends in Databases, pp. 267-402, 2011.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] “Google trends explorers” http://www.google.com/trends/explore#cmpt=q.

[8] K. Kunanusont and J. Chongstitvatana, “Finding a set of high-frequency queries for high-frequency-query-based filter for similarity join” in Proceedings of 12th International Conference on Electrical Engineering/ Electronics, Computer, Telecommunications and Information Technology (ECTI-CON), 2015.

[9] 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.

[10] “UCI Machine Learning Repository” https://archive.ics.uci.edu/ml/ datasets.html Obtained on 8th January 2015.

[11] http://www.cs.berkeley.edu/~jnwang/codes/adapt.tar.gz.