Parallel Similarity Join with Data Partitioning for Prefix Filtering

Jaruloj Chongstitvatana[1] and **Methus Bhirakit**[2], Non-members

ABSTRACT

Similarity join is necessary for many applications, such as text search and data preparation. Measuring the similarity between two strings is expensive because inexact match is allowed and strings in databases are long. To reduce the cost of similarity join, the filtering-and-verify approach reduces the number of string pairs which require the computation of the similarity function. Prefix filtering is a filter-and-verify method that filters out dissimilar strings by examining only their prefixes. The effectiveness of prefix filtering depends on the length of the examined prefix. An adaptive method is proposed to find a suitable prefix length for filtering. Based on this concept, we propose to divide a dataset into partitions, and a suitable prefix length is determined for each partition. It also allows similarity join to run in parallel for each data partition. From our experiment, the proposed method achieves higher performance because the number of candidates is reduced and the program can execute in parallel. Moreover, the performance of the proposed method depends on the number of data partitions. If the data partition is too small, the chosen prefix length for each partition may not be optimal.

Keywords: Similarity Join, Prefix Filtering, Parallel Join, Data Partitioning

1. INTRODUCTION

String data are present in many applications such as digital libraries, web content and social network applications. The cost of searching for string data in a database is high because of many factors. First, string data from different sources can be in different word order and use different word forms, or can be misspelled. Various degree of mismatch must be considered. Second, traditional index structures such as B+ tree and kDB tree, which are used to accelerate the search for basic data types, do not support string data.

To cope with varied word forms, different measures for string similarity are proposed. One type of measures is based on characters in strings, such as edit distance [1], which counts the number of operations on characters that make two strings equal. Many efficient algorithms such as [2] are proposed to reduce the computation time of the edit distance. Another type of measures maps different forms of the same word to the same token, and uses a set of tokens to represent a string. Then, string similarity is measured from the number of common tokens, the number of different tokens and the number of tokens in each string. This type of measures includes overlap similarity, cosine similarity, and Jaccard similarity [3]. The computation cost for string similarity is high, especially when strings are long. The filter-and-verify approach is adopted to reduce the cost of finding similar strings [4] by filtering out some dissimilar strings first and verifying the remaining candidate strings using a similarity function.

Because a large number of strings need to be examined in the filter step, index structures are created to speed up the process. Tries are mostly used as indices when string similarity is measured by characters. An advantage of tries as indices is that a common prefix of strings can be organized together. In [5], tries are used to group strings with common prefixes together, and thus reduce the cost of searching. On the other hand, inverted lists are mostly used for set-based approach. In [6], strings which contain the same token are grouped together in an inverted list, and inverted lists of all tokens in the database are stored in an inverted index. Binary search is used to efficiently locate the required strings in inverted lists. However, it still takes a long time when the database contains a large number of strings and these strings are long. Prefix filtering, proposed in [4], examines only some prefix of, but not the whole, strings to eliminate non-candidate strings for a query. An index structure is also built to store strings containing the same tokens. In prefix filtering, when longer prefixes are examined, more candidate strings can be eliminated with the cost of filter time. The adaptive framework for similarity join [7] chooses a suitable prefix length for filtering by comparing the time increased in the filter step against the time saved in the verify step.

Because the cost effectiveness of prefix filtering depends on the length of prefixes that are examined in the filter step, it is crucial to find an optimal prefix length for a dataset. When a prefix length is chosen, it is used in the query of all data records. However, some subsets of data records may require

Manuscript received on October 30, 2014.
Final manuscript received December 5, 2014.

[1] The author is with Department of Mathematics and Computer Science, Faculty of Science, Chulalongkorn University, Bangkok 10330, Thailand., E-mail: jaruloj@gmail.com

[2] The author is with Sirimedia Co., Ltd., Sathorn, Bangkok 10550, Thailand., E-mail: methusb@gmail.com

shorter prefixes, and other different subsets of data may require longer ones. In this paper, we partition a dataset into smaller subsets and find a prefix length which is suitable for each subset. One advantage of this method is that the computation time can be reduced because some data partitions need shorter prefix length for filtering. Another advantage is that this method lends itself naturally to parallel implementation because filtering can be done independently on each data partition. The necessity of parallel join algorithms is evident from many researches such as [8, 9, 10, 11]. Furthermore, there is only small overhead to parallelize the prefix filtering algorithm because the dataset can be partitioned arbitrarily. We implement the proposed algorithm and perform an experiment to compare its performance with ppjoin, ppjoin+ [6] and AdaptJoin [7]. It is shown that the proposed algorithm run faster, especially when the similarity threshold is low, i.e. less similar strings are allowed.

In this paper, section 2 describes related works, including similarity measure, similarity join and parallel join. Section 3 explains the proposed method - prefix filtering with data partitioning. To compare with existing algorithms, experiments are performed and the results are analysed in section 4. In section 5, the conclusion and future works are discussed.

2. RELATED WORK

To eliminate nearly-duplicated string data in databases, it is necessary to search for similar strings. In this section, we first describe how string similarity is measured. Then, algorithms for similarity join, which are used to efficiently match similar strings, are described. Last, parallel join algorithms are explained.

2.1 Similarity Measure

Similarity measures for string data can be grouped into two types, which are edit-based measures and set-based measures [12]. For edit-based measures, the similarity between two strings is measured by the number of operations on characters which transform one string to the other. An example of edit-based similarity measure is edit distance which counts the number of insertions and deletions of characters in strings that makes two strings equal. For example, the words *'colour'* and *'color'* are different by one because of one deletion of *'u'* the word *'colour'* is needed. Many variations of edit distance such as [13, 14] are proposed for different purposes. However, edit distance does not work well for word re-ordering and mismatching substrings. For example, the edit distance between the strings *'cost of computing'* and *'computing cost'*, which have the same meaning, is high. From this example, the word order in the strings is of minor importance. To avoid this problem, set-based measures are used.

For set-based measures, each string or text document is tokenized and represented by a set of tokens. Words of the same meaning, such as *colour/color* and *compute/computing/computation*, are mapped to the same token. Then, the similarity between text documents r and s, denoted by $\mathrm{sim}(r, s)$, is measured by a similarity function. Various similarity functions, such as overlap similarity, Jaccard similarity, dice similarity and cosine similarity, are used for set-based approaches. Overlap similarity is measured by the number of common tokens between the two strings. Jaccard similarity is the ratio between the number of common tokens in the two strings and the total number of tokens in both strings. For dice similarity, the sum of the cardinality of the two token sets is used, instead of the total number of tokens in the two strings. Cosine similarity is defined as the ratio between the number of common tokens in the two strings and the square root of the product of the cardinality of the two token sets. The strings r and s are considered to be similar if $sim(r, s)$ is higher than a given similarity threshold T.

To compute a similarity function, it is necessary to find the number of common tokens between two strings. In text databases, strings are long and there are a large number of tokens in a string. As a result, finding common tokens between two strings takes a lot of computation.

Index structures and access methods are designed to support similarity search when set-based similarity functions are used. Similarity search and similarity join are described next.

2.2 Similarity Join

Similarity join searches for pairs of similar strings r and s from relations R and S. A string is tokenized, and it is represented by the set of tokens in the string. Consider a text database S containing the strings w_1, w_2 and w_3, where w_1 contains the words join, similarity, distance, inverted, and index, w_2 contains the words join, distance, index, spatial, and tree and w_3 contains the words similarity, distance, index, Euclidean, spatial, and tree. Each word is represented by a token. Tokens $t_1, t_2, t_3, t_4, t_5, t_6, t_7$ and t_8 represent the words join, similarity, distance, inverted, index, Euclidean, spatial, and tree, respectively. In this database, the strings w_1, w_2 and w_3 are represented by the sets of tokens $\{t_1, t_2, t_3, t_4, t_5\}$, $\{t_1, t_3, t_5, t_7, t_8\}$ and $\{t_2, t_3, t_5, t_6, t_7, t_8\}$, respectively.

As mentioned earlier, it is costly to compute a similarity function. More efficient methods are proposed. [15] presents indexing techniques and algorithms based on inverted indices and different weighted set-based similarity functions and it estimates the similarity of records by estimating the similarity of their token sets.

The filter-and-verify approach aims to reduce the cost of similarity join by filtering out dissimilar strings first. After that, the similarity function is computed for the remaining candidate pairs of strings in the verify step. To improve efficiency of the filter step, an inverted index of the strings in the database is built and then the upper bound of the string similarity is calculated. The efficiency of this approach critically relies on the pruning power of the filters. If more strings can be eliminated, less work is needed in the verify step.

Prefix filtering [4] reduces the cost of comparing every pair of tokens by reducing the number of token comparisons. Tokens in each string are sorted by the frequency of their occurrences in the database. In other words, in a list of tokens that represents a string, a token which appears with lower frequency comes before one with higher frequency. To find at least t common tokens for a string s, the p-prefix of s, which is the set of first $|s|-t+p$ tokens of s, must be compared. Indices, which are lists of strings containing each token in their prefixes of different length, are stored. Tokens in the p-prefix of the query string are compared to each record in the index. If the p-prefix contains no common token with the query, the string is not a candidate answer for the query and can be filtered out. It is a challenge to find the suitable prefix length for each query. The longer prefixes give higher accuracy, but lead to higher computation cost. Furthermore, different prefix lengths are appropriate for different query objects.

Xiao et al. [16] proposes an algorithm for similarity join by considering the suffix tokens of the record to increase the pruning power. It is shown that this method can prune the data sets far better than Ed-join [17]. Furthermore, the prefix filtering technique is analyzed and an optimized global ordering over the token universe is proposed.

Since the performance of prefix filtering depends on the length of prefix used in the filtering, Wang et al. [7] proposes an adaptive framework for similarity join which dynamically chooses a prefix length for each query to reduce the processing. An optimal prefix length is chosen, based on the trade-off between the filter cost and the verify cost. It is necessary to estimate the filter cost and the verify cost, and a cost model for the estimation is proposed. The index for a database contains delta inverted lists $\Delta I_k, fork = 1, 2, \ldots, n$. A delta inverted list ΔI_k is the set of inverted lists $\Delta I_k(t_i)$ for all tokens t_i, and $\Delta I_k(t_i)$ contains any string s such that ti is in the k-prefix of s, but not in the $(k-1)$-prefix. These indices store the objects which have the given token in their prefix of that length, but not in any shorter prefix to reduce indexing time and space. In the cost model, the total cost for a prefix length scheme is the filtering cost together with the verification cost. The filtering cost of l-prefix length scheme can be estimated by summing the lengths of all delta inverted lists of the 1-prefix to l-prefix. The cost of verification is estimated from the average cost to verify an object and the size of candidate sets obtained from the filter step. The prefix length l is chosen if the estimated total cost for l-prefix scheme is lower than the cost for $(l+1)$-prefix scheme. The accuracy of the cost estimation depends on the number of sampled objects.

Because of large databases used presently and the availability of parallel processors, parallel join algorithms are also adopted to reduce the processing time.

2.3 Parallel Join

Various parallel algorithms for similarity join have been studied. ε-k-d-B trees are proposed in [8] for multidimensional data. An ε-k-d-B tree is built for each data partition and allocated to each processor. When the necessary data is not stored locally for a processor, the data must be exchanged between processors. A cost model is developed to dynamically set the threshold of the leaf size. Furthermore, data partitioning strategies are proposed for multidimensional data.

[9] proposes a parallel join algorithm for graphics processing units (GPU). A multi-dimensional dataset is mapped into a set of space-filling curves, and similarity join can be solved by sorting and searching on these curves. Simple operations on GPU can be used effectively for this method because of the problem transformation. The efficiency of this method is resulted from the size of the interval searches.

Although there are some researches on parallel algorithms for similarity join, there have been only few studies on parallel set-based similarity join. Vernica et al. [10] adopts similarity join query in Google's MapReduce [11] framework. A method to sort tokens in MapReduce framework is proposed. ppjoin+ [6] is used to generate candidate pairs of similar strings because of its low memory utilization. Data partitioning and controlling based on the length filter are also described.

3. PROPOSED METHOD

In [7], it is shown that the suitable prefix length for a query depends on the token set of the query objects. Similarly, different prefix lengths yield optimum performance for different partitions of a database. In this paper, we propose to improve the performance of the prefix filtering by choosing different prefix lengths for different partitions of data in a dataset. The dataset is divided into partitions, and the adaptive framework for similarity join is used to find an optimized prefix length for each partition, as shown in Figure 1. This can reduce the overall cost of similar join. Furthermore, the algorithm can be parallelized. Figure 1 compares the adaptive framework with our proposed method.

For our proposed method, the indices are created in the initial step, and these indices are used for

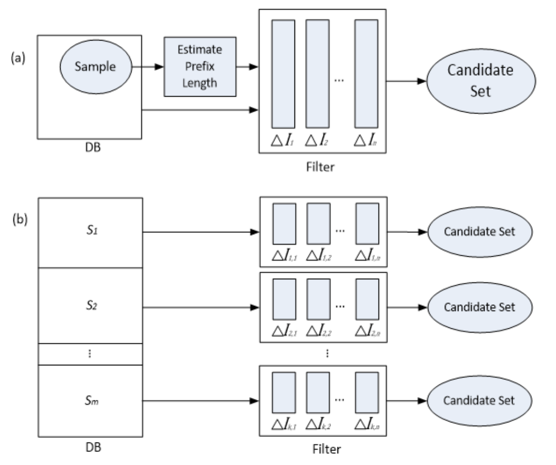


Fig. 1: *The comparison between adaptive framework (a) and our proposed method (b).*

queries as shown in the filter step. In the initial step, the database is randomly divided into partitions of approximately-equal size. Then, a delta inverted index is built for each partition, using the same method as in [7], as shown below.

Initial Step:

1. Sort tokens in each string by their frequency of occurrences in the database.
2. Randomly divide the dataset S into m partitions, denoted by $S_1, S_2, \ldots,$ and S_m, so that each partition contains at most $\lceil |S|/m \rceil$ records.
3. Build the indices $I_1, I_2, \ldots,$ and I_m, where I_k contains only the inverted list of each token in S_k.

In the filter step, a thread is created for each data partition to parallelize the algorithm, and the query object is assigned to all threads. Then, each thread can work on its own data partition. This involves small overhead for parallelization in comparison with existing parallel joins such as [8, 10]. For each thread, we compute the estimated cost of join using l-prefix, which is the sum of the filter cost and the estimated verification cost. The filter cost is the number of common tokens between the query object and data in ϕl, which is the set of inverted list of strings containing each token in l-prefix. ϕl is created from the union of delta inverted indices for i-prefix where i does not exceed l. The verification cost is estimated by the product of the number of candidates that are retrieved from ϕl and the average length of all strings in the database. To estimate the number of candidates in $\Delta I_{i,k}(r)$, a number of records are randomly sampled from $\Delta I_{i,k}(r)$ and check if each of them is a candidate. The number of candidates in $\Delta I_{i,k}(r)$ is estimated by the ratio between the number of all data and the number of the sampled data times the number of candidates in the sample set. We determine the prefix length by the comparison of the total cost of l-prefix and $(l-1)$-prefix. If the cost of l-prefix is more than cost of $(l-1)$-prefix, then the iteration is stopped. But if the cost of l-prefix is less than cost of $(l-1)$-prefix, this iteration would be repeated for longer prefix length. The filter algorithm is described below.

Filter Step:

```
Let $cost(l,k)$ be the cost of filter and verify the query
object for $l$-prefix length of the index $I_k$.
Partition data into $m$ partitions.
Create $m$ threads for $m$ partitions.
FOR THREAD $k = 1$ TO $m$
{       Calculate $cost(1,k)$.
        Calculate $cost(2,k)$.
        SET $l = 1$.
        WHILE ( $cost(l,k) > cost(l+1,k)$)
        {Calculate $cost(l+2,k)$.
         $l = l + 1$.
        }
        Find all strings from $I_l$ which share $l$
        more than common tokens with the
        query string.
}
```

In our method, the query object joins data in each partition simultaneously. Different prefix lengths may be optimal for the same query object in different data partitions. For self-join, we join the objects in n^{th} partition with data in all $(n+k)^{th}$ partitions, where k is a positive integer, in order to avoid the repeated work.

In the next section, the performance evaluation of our proposed method against the existing filter-and-verify methods in [5] and [9] is described.

4. EXPERIMENTS AND RESULTS

In this section, the proposed algorithm is evaluated against the following existing methods: *ppjoin*, *ppjoin+* [6] and *AdaptJoin* [7]. All algorithms are implemented in C++ and compiled using G++ with -O3 flag. Google's dense_hash_map has been used to collect overlap values for candidates. In the experiment, we use Linux machine with 4 virtual cores and 34.2 GB memory in Amazon Elastic Compute Cloud (Amazon EC2). The proposed algorithm is implemented with four threads because a 4-core processor is used in the experiment. DBLP [18] - a research publication's list of titles and authors dataset - is used as experiment data. This dataset contains 1,385,925 records and 1,000 queries. Jaccard similarity is used as a similarity function.

First, the performance, i.e. the running time, of the proposed method is compared with *ppjoin*, *ppjoin+* and *AdaptJoin*. Then, we study how the number of data partitions affects the performance of the proposed method. The running time and the number of candidate strings are measured for different numbers of data partitions.

4.1 Performance Comparison With Existing Methods

In this experiment, the proposed prefix filtering with data partitioning (*Partition*), *ppjoin*, *ppjoin+* and *AdaptJoin* are evaluated on DBLP dataset, which contains a set of data and a set of query strings. The query string q with the similarity threshold T returns all strings r such that $sim(q, r)$ is higher than the threshold T, and the similarity threshold is varied from 0.5 to 1.0 in this experiment. The dataset is partitioned into four partitions as it will be shown in the next subsection that the performance of our algorithm is best with four data partitions.

Figure 2 shows the running time of our proposed algorithm, *ppjoin*, *ppjoin+* and *AdaptJoin*. When the threshold is high, the running time of all four algorithms, except *AdaptJoin*, is low. The running time of the proposed method is not much lower than other method because the prefix length used for filtering with high similarity threshold is already short and the partitioning cannot significantly reduce the comparison. When the threshold is low, the proposed algorithm outperforms *ppjoin*, *ppjoin+* and *AdaptJoin*. The proposed algorithm can reduce the running time because it can filter out dissimilar strings effectively and can also run in parallel for each partition of data. Moreover, the result also shows the running time of the proposed method increases more slowly with the decreasing threshold.

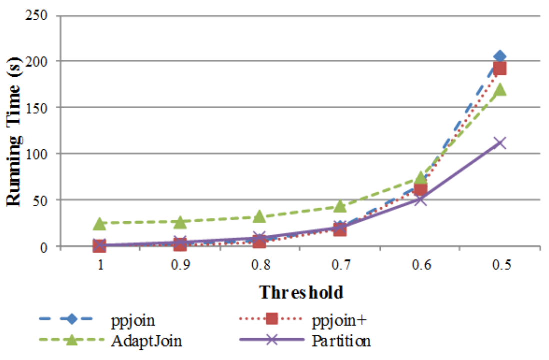


Fig.2: The running time of our algorithm, ppjoin, ppjoin+ and AdaptJoin.

4.2 Running Time With Varying Numbers of Partitions

Next, the effect of the number of data partitions on the running time of the proposed method is studied. Figure 3 shows the running time of our proposed algorithm when the database is partitioned into two to five partitions and the query threshold is varied from 0.5 to 1.0. The running time of the proposed algorithm decreases when the number of partitions increases from one to four partitions, but the running time slightly increases after that point. This is because the experiment is performed on a 4-core processor. It should be observed that the increase in the number of data partitions does not always improve the performance of the proposed method.

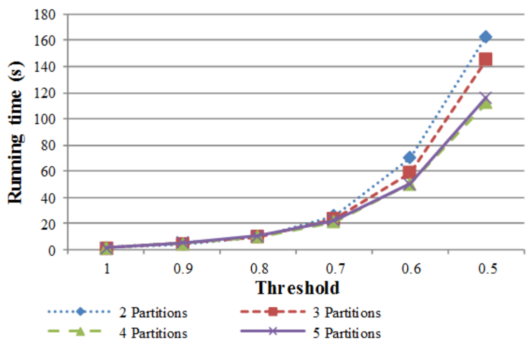


Fig.3: The running time of our algorithm with different numbers of data partitions.

To further understand the improvement the performance difference with different number of data partitions, the number of candidate strings is examined.

4.3 The Number of Candidates With Varying Numbers of Partitions

In this experiment, we examine the number of candidate strings retrieved from the proposed method with different number of partitions. Figure 4 shows the number of candidates retrieved with threshold 0.7 with different number of data partitions. We see that the number of candidates decreases with the increasing number of partitions. This is because the proposed algorithm chooses the prefix length dynamically and adjusts the prefix length for each data partition. But the number of candidates increases again when the partition number is greater than 12. This significant increase can be resulted from the size of each data partition. When a data partition is small, the cost model, which is calculated from a sample set of data, can be inaccurate and the chosen prefix length may not be optimal.

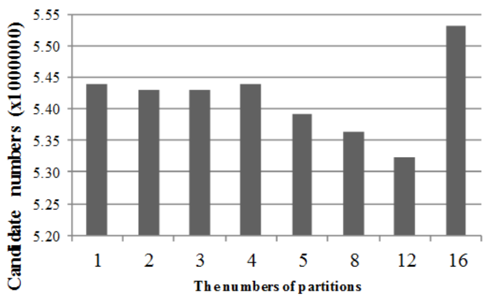


Fig.4: The number of candidates obtained from the proposed algorithm with varying numbers of partitions.

Furthermore, when the number of data partitions is varied, the decrease in the running time is resulted from both the decrease in the number of candidate strings and the parallel execution. When the number of data partitions increases from four to five, the number of candidate strings slightly decreases as shown in Figure 4, but the running time slightly increases as shown in Figure 3. The CPU utilization is also another factor for the performance.

4.4 CPU Utilization With Varying Numbers of Partitions

In this experiment, we examine the CPU utilization of the proposed algorithm. Figure 5 shows the percentage of CPU utilization of the proposed method on the dataset with threshold 0.8, with varying number of data partitions. The CPU utilization increases when the number of partitions increases up to 4 partitions, and it does not significantly change after that. However, our method cannot maximize CPU utilization because it uses only single core for index construction.

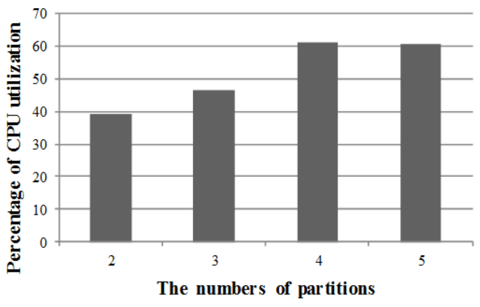


Fig.5: *CPU utilization with different numbers of data partitions.*

From the experimental result, we will discuss factors that might affect the performance of the proposed algorithm and how to improve this method.

4.5 Discussions

In the experiment, the dataset is partitioned randomly. Random partition is adopted because the cost of random partition is low. Some other clustering criteria may improve the performance of the proposed method for some datasets. However, it also brings the partition cost up.

The experiment shows that our proposed method outperforms existing algorithms. However, the performance of this method depends on the number of data partitions. When the number of data partitions increases, the number of data in each partition decreases. If the number of data in a partition is too small, the cost model used in determining the prefix length is not accurate and the running time can increase. This situation does not occur for large dataset.

Since the number of threads created in our method varies with the number of data partitions, the overhead of creating threads increases with the number of data partitions. Furthermore, the number of threads that gives the best performance also depends on the number of processors. Thus, the number of processors must be considered when we choose the number of data partitions.

5. CONCLUSION

In this paper, we present a method for similarity join based on data partitioning. Data partitioning can reduce the running time of similarity join because the prefix length is chosen for each partition appropriately. Thus, shorter prefix is used for some data partitions and the running time can be reduced. This algorithm can be naturally parallelized. We run experiments to compare the performance of our algorithm to *ppjoin*, *ppjoin+*, and *AdaptJoin*, based on DBLP dataset. It is found that our method outperforms these three algorithms for similarity join, especially when the query threshold is low. The number of data partitions chosen for our method is crucial to the performance. Factors need to be considered when the number of data partitions is chosen are the number of processors used for parallel join and the size of each data partition. The number of data partitions should correspond with the number of processors. Furthermore, each partition must not be so small that the filter-and-verify cost model becomes inaccurate. For the future work, the proposed method should be experimented on other parallel architectures, and issues of memory allocations should be explored to further improve the performance of the algorithm.

References

[1] V. I. Levenshtein, "Binary codes capable of correcting deletions, insertions, and reversals," *Soviet Physics Dokl.*, 1965, pp. 707-710.

[2] H. Bunke, and J. Csirik, "An improved algorithm for computing the edit distance of run-length coded strings," *Information Processing Letters*, Vol. 54, pp.93-96, 1995.

[3] P. Jaccard, "Distribution de la flore alpine dans le bassin des Dranses et dans quelques régions voisines," *Bulletin de la Société Vaudoise des Sciences Naturelles*, Vol. 37, pp. 241-272, 1901.

[4] S. Chaudhuri, V. Ganti, and R. Kaushik, "A primitive operator for similarity joins in data cleaning," *Proceedings of International Conference on Data Engineering (ICDE)*, pp. 5-16, 2006.

[5] J. Wang, G. Li and J. Feng, "Trie-join: efficient trie-based string similarity joins with edit," *PVLDB*, pp. 1219-1230, 2010.

[6] S. Sarawagi and A. Kirpal, "Efficient set joins on similarity predicates," *Proceedings of ACM*

Management of Data (SIGMOD), pp. 743-754, 2004.

[7] J. Wang, G. Li and J. Feng, "Can we beat the prefix filtering? An adaptive framework for similarity join and search," *Proceedings of ACM Management of Data (SIGMOD)*, pp. 85-96, 2012.

[8] K. Alsabti, S. Ranka and V. Singh, "An efficient parallel algorithm for high dimensional similarity join," *IPPS: 11th International Parallel Processing Symposium*, pp. 556-560, 1998.

[9] M. D. Lieberman, J. Sankaranarayanan and H. Samet, "A fast similarity join algorithm using graphics processing units," *Proceedings of the 24th IEEE International Conference on Data Engineering*, pp. 1111-1120, 2008.

[10] R. Vernica, M. J. Carey and C. Li, "Efficient parallel set-similarity joins using MapReduce," *Proceedings of ACM Management of Data*, pp. 495-506, 2010..

[11] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters," *Proceedings of 6th Symposium on Operating System Design and Implementation (OSDI)*, pp. 137-150, 2004.

[12] J. Wang, G. Li, and J. Fe, "Fast-join: An efficient method for fuzzy token matching based string similarity join," *IEEE International Conference on Data Engineering (ICDE)*, pp. 458-469, 2011.

[13] R. W. Hamming, *Coding and Information Theory*, Prentice-Hall, 1980.

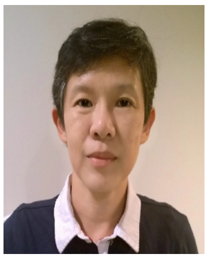
[14] F. P. Miller, A. F. Vandome, and J. McBrewster, Levenshtein Distance: *Information Theory*, Alpha Press, 2009.

[15] M. Hadjieleftheriou and D. Srivastava, "Weighted set-based string similarity," *IEEE Data Engineering Bulletin*, Vol. 33, pp. 25-36, 2010.

[16] C. Xiao, W. Wang, X. Lin and J. X. Yu, "Efficient similarity joins for near-duplicate detection", *ACM Transactions on Database Systems*, pp. 15-54, 2011.

[17] C. Xiao, W. Wang and X. Lin, "Ed-join: an efficient algorithm for similarity joins with edit distance constraints", *PVLDB*, pp. 933-944, 2008.

[18] "The DBLP Computer Science Bibliography", http://www.informatik.uni-trier.de/ ley/db/, 2012.



Jaruloj Chongstitvatana received the B.Eng. degree in computer engineering from Chulalongkorn University, Bangkok, Thailand, in 1986 and the M.Sc. and Ph.D. degrees in computer science from Michigan State University in 1989 and 1999, respectively. She is a lecturer in the Department of Mathematics and Computer Science, Faculty of Science, Chulalongkorn University. Her research interests include index structures and data management.

Methus Bhirakit received the B.S. degree (honors) in computer science from Chulalongkorn University, Bangkok, Thailand, in 2013. He is working at Sirimedia, Co., Ltd., Thailand. His research interests include index structures and data management.