Effective Early Termination Techniques for Text Similarity Join Operator*

Selma Ayse Özalp[1], and Özgür Ulusoy[2]

[1]Department of Industrial Engineering, Uludag University, 16059 Gorukle Bursa, Turkey
ayseozalp@uludag.edu.tr
http://www20.uludag.edu.tr/~ayseozalp
[2]Department of Computer Engineering, Bilkent University, 06800 Bilkent Ankara, Turkey
oulusoy@cs.bilkent.edu.tr

Abstract. Text similarity join operator joins two relations if their join attributes are textually similar to each other, and it has a variety of application domains including integration and querying of data from heterogeneous resources; cleansing of data; and mining of data. Although, the text similarity join operator is widely used, its processing is expensive due to the huge number of similarity computations performed. In this paper, we incorporate some short cut evaluation techniques from the Information Retrieval domain, namely Harman, quit, continue, and maximal similarity filter heuristics, into the previously proposed text similarity join algorithms to reduce the amount of similarity computations needed during the join operation. We experimentally evaluate the original and the heuristic based similarity join algorithms using real data obtained from the DBLP Bibliography database, and observe performance improvements with continue and maximal similarity filter heuristics.

1 Introduction

The text similarity join operator, as its name implies, joins two relations if their join attributes, which consist of pure text, are highly similar to each other. The similarity between join attributes is determined by well-known techniques such as *tf-idf* weighting scheme [1] and *cosine similarity* measure from the Information Retrieval (IR) domain. The text similarity join operator has various application domains. Cohen [2], Gravano et al. [3], and Schallehn et al. [4] use this operator for the integration of data from distributed, heterogeneous databases that lack common formal object identifiers. For instance, in two Web databases listing research institutions, to determine whether the two names "AT&T Labs" and "AT&T Research Labs" denote the same institution or not, text similarity join operator may be employed.

Meng et al. [5] use the text similarity join operator to query a multidatabase system that contains local systems managing both structured data (e.g., relational database) and unstructured data (e.g., text). As an example let's assume that we have two global relations: *applicants* containing information about job applicants and their resumes, and *positions* including the description of each job; then the text similarity join operator is

* This research is supported by a joint grant from TÜBITAK (grant no. 100U024) of Turkey and the National Science Foundation (grant INT-9912229) of the USA.

used to answer queries like "for each position, find *k* applicants whose resumes are most similar to the position's description". Jin et al. [6] employ similarity join operator for solving the problem of record-linkage in the context of data cleansing. In [7, 8], we describe similarity join operator to facilitate metadata based web querying [9].

We classify the text similarity join operators as top-k, threshold, and directional similarity join operators [8] such that the *top-k similarity join* takes two relations *R* and *S*, and an integer *k* as input, then joins tuple pairs from *R* and *S* according to the similarity of their textual join attributes, and returns *k* joined tuples having the highest similarity values. The *threshold similarity join* also takes two relations *R* and *S*, and a real *threshold value* in the range [0..1] as input, and joins tuples from *R* and *S* if the similarity of their textual join attributes is greater than or equal to the threshold value. The last similarity join operator, called *directional similarity join*, joins each tuple from relation *R* with *k* most similar tuples from relation *S*, and returns at most $|R|*k$ joined tuples where $|R|$ is the number of tuples in relation *R*. In this study, we focus on the directional similarity join operator, and we try to reduce the amount of similarity comparisons executed by employing some early termination heuristics (e.g., Harman, quit, continue, and maximal similarity filter) from the IR domain. These heuristics improve the performance of the join operation by considering only the tuple pairs that have high similarity to each other and ignoring the ones having small or no similarity. We also show through experimental evaluation that early termination heuristics improve the performance of the similarity join operator considerably in terms of the number of disk accesses made and the amount of similarity computations performed.

The rest of the paper is organized as follows. In the next section, we describe the related work. A brief summary of the previously proposed directional join algorithms and the early termination heuristics are presented in sections 3 and 4. In Section 5, we experimentally evaluate and compare all the algorithms in terms of the CPU time required for processing, the number of tuple comparisons and the number of disk accesses made. Finally, we conclude our discussion in Section 6.

2 Related Work

Recently, similarity join operator for both low and high dimensional data has become a popular research topic as it is used in variety of applications such as data integration, data cleansing, data mining, and querying. Different techniques have been used for the similarity join of low dimensional (e.g., text) and high dimensional data (e.g., multimedia, biological data). Among the text similarity join proposals, the works presented in [3, 4, 6] describe processing techniques for the threshold similarity join operator. In [2, 10, 11, 12, 13, 14], algorithms for the top-k similarity join operator are described.

Although numerous proposals exist for the threshold and the top-k similarity join operators, only Meng et al. [5] study the directional similarity join operator. They propose and experimentally evaluate three join algorithms namely, Horizontal-Horizontal Nested Loop (HHNL), Horizontal-Vertical Nested Loop (HVNL), and Vertical-Vertical Merge (VVM), which use the well-known similarity measure, *tf-idf* weighting scheme and cosine similarity measure for similarity comparisons. As the names of the algorithms imply, HHNL and HVNL algorithms are nested loops based join algorithms such that HHNL algorithm compares each document (tuple) pairs in the collections (relations), and HVNL

algorithm, on the other hand, uses the documents in one collection and the inverted file for the other collection to compute the similarities. Algorithm VVM, which is not nested loops based, uses inverted files on both collections to compute the similarities. The details of these similarity join algorithms and the early termination heuristics applied to these algorithms are given in the subsequent sections.

The similarity measure employed in [5] and also in this study is the *cosine similarity measure* with *tf-idf* weighting scheme [1] in which, each document (join attribute in the similarity join operator) is represented as a vector consisting of n components, n being the number of distinct terms (i.e., stemmed words) in the document collection, such that each component of a vector for a document gives the weight of the term i for that document. Weight of a term for a particular document is computed according to *tf-idf* value, where *tf* (term frequency) is the number of occurrences of term i within the document; and *idf* (inverse document frequency) gives more weight to scarce terms in the collection. The similarity measure is the cosine of the angle between the two document vectors such that the larger the cosine, the greater the similarity. Other measures such as Hamming distance, and longest common subsequence (LCS) for determining the similarity between short strings have also been developed. In [2, 3, 5] *tf-idf* weighting scheme and cosine similarity measure are preferred as the vector space model gives quite good matches even for short strings. Also, the vector space model allows the use of inverted indices, which makes possible for us to integrate some early termination heuristics from the IR domain during the similarity comparisons of tuples.

3 Directional Text Similarity Join Algorithms

The only study that has appeared in the literature for the directional similarity join operator were developed by Meng et al. [5] who presented three algorithms namely HHNL, HVNL, and VVM for the join operator. The HHNL (Horizontal-Horizontal Nested Loops) algorithm is a blind nested loops join algorithm, in which each tuple r in relation R is compared with every tuple in relation S, and k most similar tuples from S are joined with tuple r. In [5], the input relations R and S are read from disk. After reading X tuples from R into the main memory, the tuples in S are scanned; and while a tuple in S is in the memory, the similarity between this tuple and every tuple in R that is currently in the memory is computed. For each tuple r in R, the algorithm keeps track of only those tuples in S, which have been processed against r and have the k highest similarities with r. In the HHNL algorithm, and also in all other algorithms described in [5], a heap structure is used to find the smallest of the k-largest similarities.

The HVNL (Horizontal-Vertical Nested Loops) [5] algorithm is an adaptation of the ranked query evaluation techniques in the IR domain to the join operation. In an IR system, the aim is to find the k documents in the system which are most similar to the user query. For that purpose, most of the IR systems employ inverted files. In these systems, for each term t in the user query, the term is searched from the inverted index and the ids of documents containing term t are found. Then, the similarity calculations are performed only for those documents that have at least one common term with the user query. Algorithm HVNL is a straightforward extension of this method such that for each tuple r in R, the algorithm calculates the similarity of r to all tuples in S having at least one common term with r, and selects the k most similar tuples from S. The advantage of

HVNL algorithm is that, it does not perform similarity calculations for all tuples in S as in the case of the HHNL algorithm. In the HVNL algorithm, the *inverted file* consists of (i) an *inverted index* which includes the index term (t), the number of tuples in S containing the index term (f_t), and a pointer to its corresponding inverted list entry, and (ii) an *inverted list* which stores tuple id having the index term t, and the frequency of the term in that tuple ($f_{s,t}$). In the HVNL algorithm the inverted index is stored in the memory, the inverted list entries, and the relations R and S are read from disk.

The algorithm VVM (Vertical-Vertical Merge) employs sorted inverted indices with respect to the index terms on both of the input relations R and S [5]. The VVM algorithm scans both inverted files on the input relations at the same time. During the scan of the inverted indices, if both index entries correspond to the same index term, then similarities are accumulated between all tuples in the inverted lists of the indices. The VVM algorithm assumes that, both inverted files as well as relations R and S are read from disk. In order to store intermediate similarities between every pair of tuples in the two relations, the algorithm needs $|R|*|S|$ accumulators[1] that are stored in main memory. The strength of the algorithm is that it scans the inverted files only once to compute similarities between every pair of tuples. However, the memory requirement for the accumulator is so large that it cannot be run for relations having large number of tuples. As an example, let's assume that both relations R and S consist of 100,000 tuples, and each similarity value requires 4 bytes (size of float), so the memory allocated for the accumulator should be at least 100,000*100,000*4 bytes = 40Gb. In this study, we do not consider the algorithm VVM due to its huge memory requirement.

4 Heuristic Based Directional Similarity Join Algorithms

In the subsequent sections, we first describe early termination heuristics [15] from the IR domain that we use to improve the performance of directional similarity join operation, and then we briefly introduce directional similarity join algorithms employing these heuristics.

4.1 Harman Heuristic

Harman et al. [16] proposed a heuristic to decrease the number of similarity computations performed during the search of similar documents to a user query. We apply this heuristic to the HVNL algorithm as it employs an inverted index over the relation S. The HVNL algorithm extended with Harman heuristic is called HVNL-Harman in which, for each tuple r in relation R, weights of the terms in r are examined, and the inverted index is accessed only for these terms having a weight greater than the 1/3 of the highest weighted term in r. This heuristic is implemented by modifying the original HVNL algorithm as presented in Figure 1. The HVNL-Harman algorithm considers S tuples which have high weighted terms in tuple r, and does not perform similarity computations for other S tuples that do not contain high weighted terms.

---

[1] Accumulator is a set of real numbers (A_{rs}) each stores an accumulated similarity between tuples r and s.

```
1.  for each tuple r in R
2.    {compute weights (w_t) of each term in r and sort the terms with
       respect to w_t in descending order;
3.    for each term t in r having weight w_t > (max{w_t for all t in r}/3)
4.      if t also appears in S
5.        if the inverted file entry of t on S (I_1^t) is in the memory
6.          accumulate similarities;
7.        else (if the inverted file entry of t on S (I_1^t) is not in the memory)
8.          if the available memory space can accommodate I_1^t
9.            read in I_1^t;
10.         else
11.           find the inverted file entry in the memory with the lowest
              term frequency and replace it with I_1^t ;
12.         accumulate similarities;}
13. find the tuples in S which have the k largest similarities with r;
```

Fig. 1. HVNL algorithm with Harman heuristic (HVNL-Harman)

4.2 Quit and Continue Heuristics

Moffat et al. [17] also suggested to sort the terms in the user query with respect to their weights in descending order, and to access the inverted index with respect to this order. They place an a priori bound (i.e., accumulator bound) on the number of candidate documents that can be considered for the similarity calculation. New documents are compared until this bound is reached. The idea behind this heuristic is that, terms of high weight are permitted to contribute to the similarity computation, but terms of low weight are not. When the bound is reached; in the *quit* approach, the cosine contribution of all unprocessed terms are ignored, and the accumulator contains only partial similarity values for documents. In the *continue* strategy, documents that do not have an accumulator are ignored, but documents for which accumulators have already been created continue to have their cosine contributions accumulated. When the processing ends, the computation of full cosine values for a subset of the documents becomes completed.

As the quit heuristic allows only the partial similarity computation, it is not suitable for the directional similarity join operator. To find top-k similar tuples for a given tuple *r*, we need to have full cosine values and thus, we use the continue heuristic with the HVNL algorithm (HVNL-Continue). In this variation of the HVNL algorithm, for each tuple *r* of *R*, only *s* tuples from *S* which have high weighted terms in *r* are considered for similarity computations until the accumulator bound on the number of tuples that can be considered for similarity computations is reached. When the accumulator bound is reached, the full cosine similarities between tuple *r* and *s* tuples become computed and the k-most similar tuples to *r* are selected. In the HVNL-Continue algorithm, we need document vectors (term weights) for tuples in relation *S* to compute the full cosine similarity values. Term weights for each *s* tuple can be computed prior to the join operation by just passing over the relation only once as a one time cost. For the implementation of the HVNL-Continue algorithm, we modify the 3rd line of the algorithm in Figure 1 as "for each term *t* in (sorted terms list of) *r*". Also, in the 6th and 12th lines of the algorithm, for each tuple *r* in relation *R*, we increase the value of a counter variable by 1 each time a new *s* tuple is considered

for similarity computation, and when the value of the counter becomes equal to the predetermined accumulator bound, the for loop in line 3 is exited, and the counter is reset.

4.3 Maximal Similarity Filter Heuristic

"Maximal similarity filter" [7] is another technique that may be used to reduce the number of tuple comparisons made during the directional text similarity join operation. Let $\boldsymbol{u_s}$ = $<u_1\ u_2\ \ldots\ u_n>$ be the term vector corresponding to the join attribute of tuple s of S, where u_i represents the weight of the term i in the join attribute. Assume that the *filter vector* $\boldsymbol{f_R}$ = $<w_1\ \ldots\ w_n>$ is created such that each value w_i is the maximum weight of the corresponding term i among all vectors of R. Then, if $cos\_sim\ (\boldsymbol{u_s}, \boldsymbol{f_R}) < V_t$ then s can not be similar to any tuple r in R with similarity above V_t. The value $cos\_sim\ (\boldsymbol{u_s}, \boldsymbol{f_R})$ is called the *maximal similarity* of a record s in S to any other record r in R.

In the HVNL algorithm with maximal similarity filter (HVNL-Max-Filter), the inverted list entries are accessed with respect to descending order of maximal similarity values of s tuples. For each term t in tuple r of R, the inverted index is entered and the similarity comparisons are stopped at the point when the maximal similarity value ($cos\_sim\ (\boldsymbol{u_s}, \boldsymbol{f_R})$) for the tuple s is less than the smallest of the k-largest similarities computed so far for tuple r, since it is not possible for s to be in the top-k similar tuples list. The maximum weight of a term for a given relation is determined while creating the vectors for the tuples, and the filter vector for each relation may be formed as a one-time cost. To apply this heuristic, we need to sort the inverted list entries with respect to maximal similarity values of tuples just once during the preprocessing step. The HVNL-Max-Filter algorithm is also very similar to the HVNL-Harman algorithm (Figure 1). One difference is, the 3rd line of HVNL-Harman is changed as "for each term t in r". Also, in lines 6 and 12, similarity computations for s tuples having term t are performed if the maximal similarity value for the tuple s is greater than the smallest of the k-largest similarities computed so far for tuple r, otherwise the for loop in line 3 is exited.

We also apply the maximal similarity filter heuristic to the HHNL algorithm (i.e., HHNL-Max-Filter), in which we sort the tuples in relation S in descending order of their maximal similarity filter values as a preprocessing step, and we terminate the inner loop when the maximal similarity filter for the s tuple that is being processed is less than the smallest of the k-largest similarities computed so far for the tuple r.

5 Experimental Results

We compared the performance of HHNL, HVNL, HVNL-Harman, HVNL-Continue, HVNL-Max-Filter, and HHNL-Max-Filter in terms of the number of tuple comparisons made, the number of disk accesses required, and the CPU time needed. For the experimentation, we implemented these algorithms in C programming language under MS WindowsXP operating system. We did not include VVM since it requires huge amount of memory to keep intermediate similarities between tuple pairs. In the implementation, the relations R and S are stored on disk and each block read from the relations contain 10000 tuples. For the HVNL and its variations, the inverted index is in-memory, however the inverted list entries are stored on disk and up to 5000 inverted list entries are kept in the cache. An inverted list entry that is not in the cache is retrieved from disk by making

random disk access, and when the cache is full, the entry for the term having the least term frequency is replaced with the new entry.

In the experiments, we used a real dataset that consists of the bibliographic information of journal and conference papers obtained from the DBLP Bibliography database [18]. In the implementation of the directional text similarity join, the relations *R* and *S* do not contain any common tuple, and the relation *R* consists of bibliographic information of approximately 91,000 journal papers, and the relation *S* contains bibliographic information of 132,000 conference papers. The paper title attribute is chosen as the join attribute, and for each journal paper *r* in relation *R*, we try to find *k* conference papers from relation *S* having the most similar titles to the title of *r*. We created the vectors and the maximal similarity filters for the join attribute of each tuple in the relations *R* and *S*, and the inverted index on relation *S* in advance as the preprocessing step. We assumed that we have enough main memory to store the inverted index and the accumulators used for similarity calculations. The experiments were performed on a PC having Pentium III 450 MHz CPU and 320 MB of main memory.

In Figure 2, the results in terms of the number of tuple comparisons (i.e., similarity computations) performed by the implemented algorithms for different *k* values are presented. As displayed in the figure, the HHNL algorithm needs to make around 12 billion comparisons for each different *k* values to join *R* and *S*, while all versions of the HVNL algorithm do less than 900 million tuple comparisons for the same join operation. HVNL, and all variations of the HVNL algorithm perform much better than the HHNL algorithm, because of the fact that these algorithms employ inverted index on the input relation *S*, and they compare similarity of tuples which are guaranteed to have a similarity value greater than 0. The HHNL algorithm, on the other hand, makes similarity computation for all tuple pairs regardless of whether the tuples contain any common term or not.

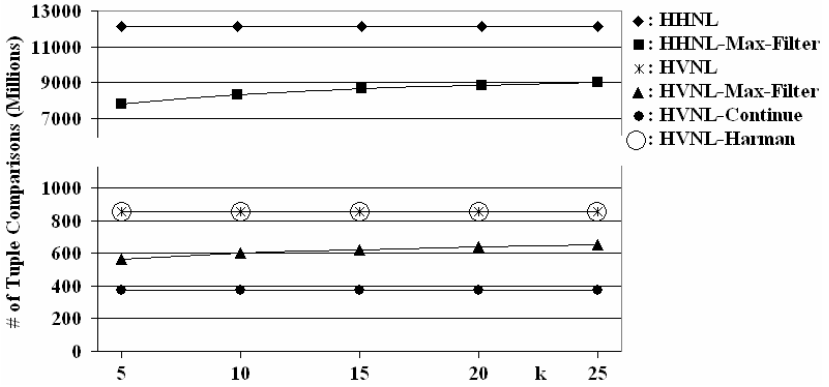

Fig. 2. Number of tuple comparisons for all algorithms vs. *k* values

The maximal similarity filter heuristic reduces the number of tuple comparison about 25% for both the HHNL and the HVNL algorithms. We use continue and Harman heuristics with the HVNL algorithm only, as these heuristics are applicable when an inverted index is employed. The continue heuristic, in which accumulator bound is set to 5000 tuples, provides more improvement on the performance of the HVNL algorithm by decreasing the number of tuple comparisons by 50%. The Harman heuristic, on the other

hand, does not improve the performance of the HVNL algorithm, because term weights for our input data are quite close to each other. Changing the value of k does not affect the number of tuple comparisons except for the maximal similarity filter heuristic. As the k value increases, maximum similarity filter heuristic needs to make more tuple comparisons to find top k similar tuples.

We also computed the number of disk accesses (Table 1) required by all algorithms when the relations R and S, and the inverted list entries on the join attribute of relation S are stored on disk. In the disk access computation, we ignored the number of disk accesses made for writing the joined tuples to the disk. According to Table 1, the number of disk accesses performed by the HHNL algorithms, which is approximately 150 disk accesses, is quite less than those obtained with the HVNL algorithms since for each term t considered during the similarity comparisons, the HVNL based algorithms read inverted list entries of term t (i.e., I_t) by making a disk access if it is not in the memory. According to the Table1, the continue heuristic reduces the number of disk accesses of the HVNL algorithm by 50%. The Harman and maximal similarity filter heuristics, on the other hand, do not lead to any reduction on the number of disk accesses required. This result is due to the fact that, the term weights in our dataset are close to each other and the Harman heuristic considers almost all terms in a tuple r during the similarity computations. The maximal similarity filter heuristic on the other hand, needs to access all the inverted list entries for all terms in a tuple r to find the s tuples having high maximal similarity values. Therefore, the maximal similarity filter heuristic only reduces the number of tuple comparisons performed when the inverted list entries are sorted with respect to the maximal similarity value of tuples.

Table 1. Number of disk accesses performed by all the algorithms for all k values

k	HHNL	HHNL-Max-Filter	HVNL	HVNL-Harman	HVNL-Continue	HVNL-Max-Filter
5-25	150	150	26282	26198	14176	26282

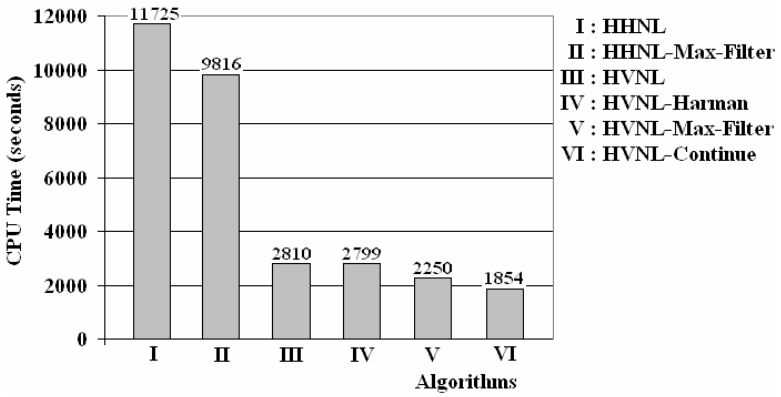

Fig. 3. CPU time required by all algorithms for the directional similarity join

Although the number of disk accesses performed by the HHNL based algorithms is quite less, the number of tuple comparisons is considerably higher than the HVNL based algorithms. To determine which group of algorithms is more efficient, we measured the

CPU time required by all of the join algorithms for k=10 and reported the results in Figure 3. As presented in Figure 3, the CPU time required to execute the join operation is 11725 seconds for the HHNL algorithm, and 2810 seconds for the HVNL algorithm, which implies that similarity computations take much longer CPU time than making disk accesses for retrieving inverted list entries. The maximal similarity filter heuristic reduces the CPU time by 16% for the HHNL and 20% for the HVNL algorithms. The continue heuristic makes 35% reduction in the processing time when the accumulator bound is set to 5000 tuples. The Harman heuristic, on the other hand, does not provide any improvement since it also does not make any reduction in the number of tuple comparisons and disk accesses.

For the continue heuristic, the accumulator bound is an important factor on the performance of the join algorithm. To show the effect of the accumulator bound on the join operation, we run the HVNL-Continue algorithm with different accumulator bounds and present the results in Table 2. We observed that, as the accumulator bound is decreased, the number of tuple comparisons falls, due to the fact that, the accumulator bound is an upper bound on the number of tuples that can be considered for similarity comparisons. The number of tuple comparisons made remain the same for different k values.

Table 2. The effect of accumulator bound for the continue heuristic

Accumulator Bound	# of Tuple Comparisons	# of Disk Accesses	CPU Time (sec)	Accuracy
5,000	372,448,481	14,176	1854	65%
10,000	604,454,778	20,001	2595	84%
15,000	732,112,934	22,678	2801	91%

We examined the accuracy of the output produced by the algorithms that employ early termination heuristics as follows: $Accuracy = |B \cap H| / |B|$, where B denotes the actual output set generated by the HHNL or HVNL algorithm, H is the output generated by the algorithm that employ any one of the early termination heuristics, and $|.|$ denotes the set cardinality. We observed that the Harman heuristic generates exactly the same output as the HHNL, and HVNL algorithms; the continue heuristic, on the other hand, could achieve 65% accuracy when the accumulator bound is set to 5000 tuples, and the accuracy can be improved up to 91% when the accumulator bound is increased to 15000 tuples. As the accumulator bound is an upper bound on the number of tuples that can be considered for the similarity comparisons, it highly affects the accuracy of the continue heuristic. For the maximal similarity filter heuristic, we observed that the accuracy of this heuristic is 100%, as it calculates the similarity for s tuples having maximal similarity value greater than or equal to the smallest of the k largest similarities computed so far for tuple r. Therefore, the heuristic considers all s tuples that can be in the result set by eliminating the ones that are not possible to be in the result.

6 Conclusion

Similarity based text join is a very useful operator to be employed in a variety of applications. In this study, we incorporate some early termination heuristics from the

Information Retrieval domain to achieve performance improvement for the text similarity join algorithms. We have demonstrated through experimental evaluation that nested loops based similarity join algorithm performs the best in terms of the number of disk accesses required; however, it compares every tuple pairs from the relations to be joined and leads to a huge amount of expensive similarity computations. Inverted index based join algorithm, on the other hand, achieves very small number of similarity computations while requiring large number of disk accesses. When we compare the processing time of the algorithms, we have demonstrated that the index based algorithm is superior to the nested loops based one, and we have observed further performance improvement by applying the maximal similarity filter and the continue heuristics to the index based join algorithm.

References

1. Salton, G.: Automatic Text Processing. Addison-Wesley (1989).
2. Cohen, W.: Data Integration Using Similarity Joins and a Word-Based Information Representation Language. ACM Trans. on Inf. Sys., Vol. 18, No. 3 (2000) 288-321.
3. Gravano, L., Ipeirotis, P. G., Koudas, N, Srivasta, D.: Text Joins in an RDBMS for Web Data Integration. In Proc. of WWW2003 (2003).
4. Schallehn, E., Sattler, K. U., Saake, G.: Efficient Similarity-Based Operations for Data Integration. Data & Knowledge Engineering, Vol. 48 (2004) 361–387.
5. Meng, W., Yu, C., Wang, W., Rishe, N.: Performance Analysis of Three Text-Join Algorithms. IEEE Trans. on Knowledge and Data Eng., Vol. 10, No. 3 (1998) 477-492.
6. Jin, L., Li, C., Mehrotra, S.: Efficient Record Linkage in Large Data Sets. In Proc. of the 8th Int. Conf. on Database Systems for Advanced Applications (DASFAA'03) (2003).
7. Özsoyoglu, G., Altingövde, I. S., Al-Hamdani, A., Özel, S. A., Ulusoy, Ö., Özsoyoglu, Z.M.: Querying Web Metadata: Native Score Management and Text Support in Databases. ACM Trans. on Database Sys., Vol. 29, No. 4 (2004) 581-634.
8. Özel, S. A.: Metadata-Based and Personalized Web Querying. PhD Thesis, Dept. of Computer Engineering, Bilkent University, Ankara (2004).
9. Özel, S. A., Altingövde, I. S., Ulusoy, Ö., Özsoyoglu, G., Özsoyoglu, Z. M.: Metadata-Based Modeling of Information Resources on the Web. JASIST, Vol. 55, No. 2 (2004) 97-110.
10. Fagin, R., Lotem A, Naor, M.: Optimal Aggregation Algorithms for Middleware. In Proc. of PODS 2001 (2001).
11. Bayardo, R. J., Miranker, D. P.: Processing Queries for First Few Answers. In Proc. of Conference on Information and Knowledge Management (1996) 45-52.
12. Chang, K. C., Hwang, S.: Minimal Probing: Supporting Expensive Predicates for Top-k Queries. In Proc. of SIGMOD 2002 (2002) 346-357.
13. Natsev, A, Chang, Y. C., Smith, J. R., Li, C. S., Vitter, J. S.: Supporting Incremental Join Queries on Ranked Inputs. In Proc. of VLDB 2001 (2001) 281-290.
14. Ilyas, I. F., Aref, W. G., Elmagarmid, A. K.: Supporting Top-k Join Queries in Relational Databases. In Proc.of VLDB 2003 (2003).
15. Vo, A. N., Krester, O., Moffat, A.: Vector-Space Ranking with Effective Early Termination. In Proc. of ACM SIGIR 2001 (2001) 35-42.
16. Harman, D. K., Candela, G.: Retrieving Records from a Gigabyte of Text on a Minicomputer Using Statistical Ranking. JASIS, Vol. 41, No. 8 (1990) 581-589.
17. Moffat, A., Zobel, J.: Self Indexing Inverted Files for Fast Text Retrieval. ACM Trans. on Inf. Sys., Vol. 14, No. 4 (1996) 349-379.
18. Ley, M.: DBLP Bibliography. At http://www.iformatik.uni-trier.de/~ley/db/ (2001).