Performance Analysis of Several Algorithms for Processing Joins between Textual Attributes*

Weiyi Meng[1], Clement Yu[2], Wei Wang[1], Naphtali Rishe[3]
[1] Dept. of Computer Science, SUNY – Binghamton, Binghamton, NY 13902
[2] Dept. of EECS, University of Illinois at Chicago, Chicago, IL 60607
[3] School of Computer Science, Florida International University, Miami, FL 33199

Abstract

Three algorithms for processing joins on attributes of textual type are presented and analyzed in this paper. Since such joins often involve document collections of very large size, it is very important to find efficient algorithms to process them. The three algorithms differ on whether the documents themselves or the inverted files on the documents are used to process the join. Our analysis and the simulation results indicate that the relative performance of these algorithms depends on the input document collections, system characteristics and the input query. For each algorithm, the type of input document collections with which the algorithm is likely to perform well is identified.
Keywords: Query processing, textual database, join, multidatabase

1. Introduction

Researches in *multidatabase system* have been intensified in recent years [1,2,6,7,9,10,13,15]. In this paper, we consider a multidatabase system that contains both local systems that manage structured data (e.g., relational DBSs) and local systems that manage unstructured data (e.g., information retrieval (IR) systems for handling text).

Because we have a database front-end, global users may submit queries that contain joins between attributes of textual type. A motivating example is presented in Section 2. A likely join comparator for textual attributes is SIMILAR_TO that matches objects with similar textual contents based on some similarity function. Since each textual object is essentially a document, the join is to pair similar documents among the two document collections corresponding to the two textual attributes.

While processing joins between non-textual attributes have been studied extensively, not much research has been reported on processing joins between textual attributes in the literature. In [3], the authors reported a case study on automating the assignment of submitted papers to reviewers. The reported study requires to match the abstract of each submitted paper with a number of profiles of potential reviewers. The problem is essentially to process a join between two textual attributes. Since the document collections involved in that study is small, efficient processing strategy of the join is not their concern. Instead, the emphasis of that work is on accuracy. A somewhat related problem is the *consecutive retrieval problem* [4,14] which is to determine, for a given set of queries Q against a set of records R, whether there exists an organization of the records such that for each query in Q, all relevant records (loosely, similar records) can be stored in consecutive storage locations. If we interpret Q and R as two document collections, then the consecutive retrieval problem deals the storage aspect of efficient retrieval of relevant documents from one collection for each document from another collection. However, a major difference between consecutive retrieval problem and the join processing problem is that the former assumes the knowledge that which documents from R are relevant to each document in Q while the latter needs to find which documents from one collection are most similar to each document from another collection. Another related problem is the processing of a set of queries against a document collection in batch. There are several differences between this batch query problem and the join problem: (1) For the former, many statistics about the queries which are important for query processing and optimization such as the frequency of each term in the queries are not available unless they are collected explicitly, which is unlikely since the batch may only be used once and it is unlikely to be cost effective to collect these statistics; (2) Special data structures commonly associated with a document collection such as an inverted file is unlikely to be available for the batch for the same reason. As we will see in the paper that the availability of inverted files means the applicability of certain algorithms. The clustering problem in IR systems [12] requires to find, for each document d, those documents similar to d in the same document collection. This can be considered as a special case of the join problem when the two document collections involving the join are identical.

This paper has the following contributions: (1) We present and analyze three algorithms for processing

*This work is supported in part by the following grants: NSF grants under IRI–9309225 and IRI–9509253, Air Force under AFOSR 93-1-0059, NASA under NAGW-4080 and ARO under BMDO grant DAAH04-0024.

joins between attributes of textual type. (2) Cost functions based on the number of I/O's for each of the algorithms are provided. (3) Simulation is done to compare the performance of the proposed algorithms. Our investigation indicates that no one algorithm is definitely better than all other algorithms in all circumstances. In other words, each algorithm has its unique value in difference situations. (4) We provide insight on the type of input document collections with which each algorithm is likely to perform well. We further give an algorithm which determines which one of the three algorithms should be used for processing a text-join. We are not aware of any similar study that has been reported before.

The rest of this paper is organized as follows. A motivating example is presented in Section 2. In Section 3, we include the assumptions and notations that we need in this paper. The three join algorithms are introduced in Section 4. Cost analyses and comparisons of the three algorithms are presented in Section 5. In Section 6, simulation is carried out to further compare the proposed algorithms and to suggest which algorithm to use for a particular situation. We conclude our discussion in Section 7.

2. A motivating example

Assume that the following two global relations have been obtained after schema integration: Applicants(SSN, Name, Resume) and Positions(P#, Title, Job_descr), where Resume and Job_descr are of type **text**. Consider the query to find, for each position, λ applicants whose resumes are most similar to the position's description. This query can be expressed in extended SQL as follows:

Select P.P#, P.Title, A.SSN, A.Name
From Positions P, Applicants A
Where A.Resume SIMILAR_TO(λ) P.Job_descr

The where-clause of the above query contains a join on attributes of textual type. This type of joins do not appear in traditional database systems. Note that "A.Resume SIMILAR_TO(λ) P.Job_descr" and "P.Job_descr SIMILAR_TO(λ) A.Resume" have different semantics. The former is to find λ resumes for each job description while the latter is to find λ job descriptions for each resume. The asymmetry of the operator SIMILAR_TO has some impact on the evaluation strategy [11].

For a given resume r and a given job description j, to be sure that r is among the λ resumes most similar to j, all resumes have to be considered. If we process the join by comparing each job description with all resumes, then after a job description d is compared with all resumes, the λ resumes most similar to d can be identified and a partial result is produced. However, if we process the join by comparing each resume with all job descriptions, then after a resume is compared with all job descriptions, no partial result can be generated. In this case, many intermediate results (i.e., similarity values between resumes and job descriptions) need to be maintained in the main memory. This observation indicates that comparing each job description with all resumes is a more natural way to process the above textual join.

Due to selection conditions on other attributes of the relations that have textual attributes, it is possible that only part of the documents in a collection need to participate in a join. For example, consider the query that is to find, for each position whose title contains "Engineer", λ applicants whose resumes are most similar to the position's description.

Select P.P#, P.Title, A.SSN, A.Name
From Positions P, Applicants A
Where P.Title like "%Engineer%" and A.Resume SIMILAR_TO(λ) P.Job_descr

If selection *P.Title like "%Engineer%"* is evaluated first, then only those job descriptions whose position title contains "Engineer" need to participate in the join.

In this paper, we are interested in studying algorithms that can be used to process the following query:

Select R1.X1, R2.Y2
From R1, R2
Where R1.C1 SIMILAR_TO(λ) R2.C2

where C1 and C2 are attributes representing two document collections (collection 1 and collection 2, respectively). Clearly, the join to be evaluated is of the form: "C1 SIMILAR_TO(λ) C2".

3. Assumptions and notations

Using the vector representation [12], each document can be represented as a list of terms together with their number of occurrences in the document. Each term is associated with a weight indicating the importance of the term in the document. Usually, terms are identified by numbers to save space. We assume that each document consists of a list of cells of the form (t#, w), called document-cell or d-cell, where t# is a term number and w is the number of occurrences of the term t in the document. All d-cells in a document are ordered in increasing order of the term number. The size of each d-cell is $|t\#| + |w|$ bytes, where $|X|$ is the number of bytes to contain X. In practice, $|t\#| = 3$ and $|w| = 2$ is sufficient. In a multidatabase environment, different numbers may be used to represent the same term in different local IR systems due to the local autonomy. Several methods may be used to overcome this problem. One method is to use actual terms rather than term numbers. The disadvantage is that the size of the document collection will become much larger (5 or more times larger). Another method is to establish a mapping between the corresponding numbers identifying the same term. An attractive method is to have a standard mapping from terms to term numbers and have all local IR systems use the same mapping. Such a standard can be very beneficial in improving the performance of the multidatabase system. It can save on communication costs (no actual terms needs to be transferred) and processing costs (it is more efficient to compare numbers than to compare actual terms or no need to search the mapping table). To simplify our presentation, we assume that the same number is always used to represent the same term in all local IR systems. Note that this assumption can be simulated by always keeping the mapping structure in the memory when different numbers are used to represent the same term in different local systems. In the remaining discussion, terms and term numbers will be used interchangeably.

Let $t_1, t_2, ..., t_n$ be all the common terms between documents D1 and D2. Let $u_1, u_2, ..., u_n$ and $v_1, v_2, ..., v_n$ be the numbers of occurrences of these terms in D1 and D2, respectively. The similarity between D1 and D2 can be defined as $\sum_{i=1}^{n} u_i * v_i$. A more realistic similarity function is to divide the similarity by the norms of the documents and to incorporate the use of the inverse document frequency weight [12], which assigns higher weights to terms which occur in fewer documents. The normalization can be carried out by pre-computing the norms of the documents, storing them and performing the divisions during the processing of the documents. The inverse document frequency weight can be pre-computed for each term and storing them as parts of the list heads in the inverted files. For the sake of simplicity of presentation, we use the number of occurrences instead of weights.

For a given term t in a given document collection C, the inverted file entry consists of a list of i-cells (short for inverted-file-cell) of the form (d#, w), where d# is a document number and w is the number of occurrences of t in the document with number d#. We assume that i-cells in each inverted file entry are ordered in increasing order of the document number. The size of each i-cell is $|d\#| + |w|$. i-cells and d-cells have approximately the same size.

We use the following notations in our discussion:

N_i — the number of documents in collection i, i=1 or 2
B — the size of the available memory buffer in pages
T_i — the number of terms in collection i
Bt_i — the size of the B+tree for collection i in pages (assume tightly packed)
p — the probability that a term in collection C1 also appears in collection C2
q — the probability that a term in C2 also appears in C1
α — the cost ratio of a random I/O over a sequential I/O
P — page size in bytes (4k)
K_i — the average number of terms in a document in collection i
J_i — the average size of an inverted file entry on collection i in pages $(5 * (K_i * N_i)/(T_i * P))$
I_i — the size of the inverted file on collection i in pages ($J_i * T_i$, assume tightly packed)
S_i — the average size of a document in collection i in pages $(5 * K_i/P)$
D_i — the size of collection i in pages ($S_i * N_i$, assume tightly packed)
I_i^t — the inverted file entry of term t on collection i
λ — operator SIMILAR_TO(λ) is used
δ — the fraction of the similarities that are non-zero

We assume that documents in each collection are stored in consecutive storage locations. Therefore, when all documents in collection i are scanned in storage order, the total number of pages read in will be D_i, which is also the total I/O cost. On the other hand, if documents in collection i are read in one at a time in random order, and a document is not kept in the memory after it is processed, then the total number of pages read in will approximately be $N_i * \lceil S_i \rceil$, where $\lceil A \rceil$ denotes the ceiling of A, and the total cost will approximately be $N_i * \lceil S_i \rceil * \alpha$, where α is the cost ratio of a random I/O over a sequential I/O due to the additional seek and rotation delay of a random read. Similarly, we assume that inverted file entries on each collection are stored in consecutive storage locations in ascending order of the term numbers and typically $\lceil J_i \rceil$ pages will be read in when an inverted file entry is brought in the memory in random order.

Note that for a given document collection, if document numbers and term numbers have the same size, its total size is the same as the total size of its corresponding inverted file.

In this paper, only I/O cost will be used to analyze and compare different algorithms as if we have a centralized environment where I/O cost dominates CPU cost.

4. Algorithms

In this section, we present three algorithms for processing joins on textual attributes. We assume that the existence of the inverted file on all document collections.

Depending on how documents and/or inverted files are used to evaluate a join, three basic algorithms can be constructed. The first algorithm is to use only documents to process the join, the second algorithm is to use documents from one collection and the inverted file from another collection to evaluate the join, and the third algorithm uses inverted files from both collections to do the same job. A collection of documents can be represented by a document-term matrix where the rows are the documents and the columns are the terms or the inverted file entries of the terms. Therefore, we name the first algorithm the **Horizontal-Horizontal Nested Loop (HHNL)**, the second algorithm the **Horizontal-Vertical Nested Loop (HVNL)**, and the third algorithm the **Vertical-Vertical Merge (VVM)**.

4.1 Algorithm HHNL

A straightforward way for evaluating the join is to compare each document in one collection with every document in the other collection. Although simple, this method has several attractive properties. First, if one or two of the collections can be reduced by some selection conditions, only the remaining documents need to be considered. Second, documents can be read in mostly sequentially resulting in mostly cheap sequential I/Os.

From the discussion in Section 2, we know that it is more natural to process the join by comparing each document in C2 with all documents in C1. That is, it is more natural to use C2 as the outer collection and C1 as the inner collection in the join evaluation. We call this order *the forward order* and the reverse order *the backward order*. The backward order can be more efficient if C1 is much smaller than C2. Due to space limitation, the backward order will not be studied further in this paper. Interested readers are referred to [11].

We adopt the policy of letting the outer collection use as much memory space as possible. The case that lets the inner collection use as much memory space as possible is equivalent to the backward order. With this memory allocation policy, the algorithm HHNL can be described as follows: After reading in the next X documents of C2 into the main memory, for some integer X to be determined, scan the documents in C1 and while a document in C1 is in the memory, compute the similarity between this document and every document in C2 that is currently in the memory. For each document $d2$ in C2, keep track of only those documents in C1 which have been processed against $d2$ and have the λ largest similarities with $d2$.

More rigorously, with C2 as the outer collection, we need to reserve the space to accommodate at least one document in C1. That is, $\lceil S_1 \rceil$ pages need to be reserved for C1. We also need to reserve the space to save λ similarities for each document in C2 currently in the memory. Assume that each similarity value occupies 4 bytes. Then the number of documents in C2 that can be held in the memory buffer of size B can be estimated as: X = $(B - \lceil S_1 \rceil)/(S_2 + 4\lambda/P)$

We now present the algorithm HHNL:

```
While (there are documents in C2 to be read in)
    If there are X1 = min{N2, X} or more unprocessed
      documents in C2 left
         input the next X1 unprocessed documents in C2
           into the main memory;
    Else input the remaining unprocessed documents in
      C2 into the main memory;
    For each unprocessed d2 of D2 in the memory
       {For each document d1 in C1
           {Compute the similarity between d2 and d1;
              If it is greater than the smallest of the λ
                largest similarities computed so far for d2
                   {replace the smallest of the λ largest sim-
                      ilarities by the new similarity;
                      update the list of the documents in C1
                      to keep track of those documents
                      with the λ largest similarities with d2;
                   } }  }
```

4.2 Algorithm HVNL

HVNL uses documents in one collection and the inverted file on the other collection to compute the similarities. In an IR system, processing a user query, which can be considered as a document, is to find the λ documents in the system which are most similar to the user query. One way to process such a query is to compare it with each document in the system. This method requires almost all entries in the document-term matrix be accessed. A more efficient way is to use the inverted file on the document collection. This method only needs to access those inverted file entries corresponding to the terms in the query. Since the number of terms in a query is usually a very small fraction of the total number of terms in all documents in the system, the inverted file based method accesses only a very small portion of the document-term matrix. HVNL is a straightforward extension of this method to the situation where we need to find the λ most similar documents from one collection for every document in another collection.

The process of using the inverted file to compute the similarities between a document d in C2 to documents in C1 can be described as follows. Let (t, w) be the next d-cell to be considered in d. Let the inverted file entry corresponding to t on C1 be $\{(d_1, w_1), ..., (d_n, w_n)\}$, where d_i's are document numbers. After t is processed, the similarity between d and document d_i as accumulated so far will be $U_i + w * w_i$, where U_i is the accumulated similarity between d and d_i before t is considered, and $w * w_i$ is the contribution due to the sharing of the term t between d and d_i, i=1,...,n. After all terms in d are processed, the similarities between d and all documents in C1 will be computed, and the λ documents in C1 which are most similar to d can be identified.

Note that before the last d-cell in d is processed, all intermediate similarities between d and all documents in C1 need to be saved. The amount of memory needed for such purpose is proportional to N_1. Further analysis can reveal that using the inverted file on C2 to process the join needs more memory space to store intermediate similarities (the amount is proportional to $\lambda * N_2$). In practice, only non-zero similarities need to be saved. We use δ to denote the fraction of the similarities that are non-zero, $0 < \delta < 1$.

HVNL will read in each document d in C2 in turn and while d is in the memory, all inverted file entries on C1 corresponding to terms in d will be read in to process d. Note that not all terms in d will necessarily appear in C1. To reduce the I/O cost, inverted file entries that are read in for processing earlier documents are kept in the memory to process later documents. Due to space limitation, usually not all inverted file entries read in earlier can be kept in the memory. Therefore, a policy for replacing an inverted file entry in the memory by a new inverted file entry is needed. Let the frequency of a term in a collection be the number of documents containing the term. This is known as *document frequency*. Document frequencies are stored for similarity computation in IR systems and no extra effort is needed to get them. Our replacement policy chooses the inverted file entry whose corresponding term has the lowest frequency in C2 to replace. This reduces the possibility of the replaced inverted file entry to be reused in the future. To make the best use of the inverted file entries currently in the memory, when a new document $d1$ in C2 is processed, terms in $d1$ whose corresponding inverted file entries are already in the memory are considered first. A list that contains the terms whose corresponding inverted file entries are in the memory will be maintained.

We now present the algorithm HVNL:

```
For each document d in C2
    {For each term t in d
        If t also appears in C1
            {Read in the inverted file entry of t on C1
               (I_1^t) if it is not already in;
             Accumulate similarities;
            }
      Find the documents in C1 which have the λ largest
        similarities with d;
    }
```

For each inverted file, there is a B+tree which is used to find whether a term is in the collection and if present where the corresponding inverted file entry is located.

One possible way to improve the above algorithm is to improve the selection of the next document to process. Intuitively, if we always choose an un-processed document in C2 whose terms' corresponding inverted file entries on C1 have the largest intersection with those inverted file entries already in the memory as the next document to be processed, then the likelihood of an inverted file entry already in the memory to be reused can be increased. This seemingly attractive alternative has two potential problems. First, by not reading in documents in their storage order, more expensive random I/Os will be incurred. Second, we have the following proposition:

Proposition: The problem of finding an optimal order of documents in C2 so that the best performance can be achieved is NP-hard.

Sketch of Proof: It was shown in [8] that the following problem known as the *Optimal Batch Integrity Assertion Verification* (OBIAV), which is to find an optimal order for verifying a set of integrity constraints and verifying each such constraint requires a set of pages be brought in from secondary storage to the memory, is an NP-hard problem. This problem can be reduced to the optimal order problem in our case and vice versa. Therefore, the optimal order problem in our case is also NP-hard. ∎

4.3 Algorithm VVM

Algorithm VVM uses inverted files on both collections to compute the similarities. The strength of this algorithm is that it only needs to scan each inverted file once to compute similarities between every pair of documents in the two collections regardless of the sizes of the two collections provided that the memory space is large enough to accommodate intermediate similarity values. In this case, VVM can be at least as good as HHNL because HHNL needs to scan each document collection at least once and the size of the inverted file on a collection is about the same as the size of the collection itself. VVM tries to compute similarities between every pair of documents in the two collections simultaneously, as a result, it needs to save the intermediate similarities. Thus, the memory requirement for saving these similarities is proportional to $N_1 * N_2$, which can be so large such that VVM can not be run at all. In summary, VVM is likely to perform well for document collections that are large in size (such that none can be entirely held in the memory) but small in number of documents. This is possible if each document has a large size.

Algorithm VVM can be described as follows: we scan both inverted files on the two collections. During the parallel scan, if two inverted file entries correspond to the same term, then invoke the similarity accumulating process.

Recall that we assumed that inverted file entries are stored in increasing order of the term numbers. Therefore, one scan of each inverted file is sufficient (very much like the merge phase of sort merge). The similarity accumulating process can be described as follows. Let $I_1^t = \{(r_1, u_1), ..., (r_m, u_m)\}$ and $I_2^t = \{(s_1, v_1), ..., (s_n, v_n)\}$ be two inverted file entries for the same term t on the two collections, respectively. After the two inverted file entries are processed, the similarity between documents r_p and s_q as accumulated so far will be $U_{pq} + u_p * v_q$, where U_{pq} is the accumulated similarity between r_p and s_q before t is considered, $p = 1, ..., m, q = 1, ..., n$.

We can extend the above algorithm VVM as follows to tackle the problem of insufficient memory space for all intermediate similarities. Suppose SM is the total number of pages needed to store the intermediate similarities when all pairs of documents in the two collections are considered at the same time. Suppose M is the available memory space for storing the intermediate similarities. If SM > M, divide collection C2 into $\lceil SM/M \rceil$ subcollections and then compute the similarities between documents in each subcollection and documents in C1, one subcollection at a time. Since for each such subcollection, one scan of the original inverted files on both collections is needed, this extension incurs a cost which will be $\lceil SM/M \rceil$ times higher than that when the memory is large enough to hold all intermediate similarities. For a more detailed cost analysis, see Section 5.3.

5. I/O cost analysis

5.1 Algorithm HHNL

Let X be the number of documents in C2 that can be held in the memory buffer of size B as defined in Section 4.1. Since for each X documents in C2, C1 needs to be scanned once, the total I/O cost of HHNL can be estimated as below:

$$\text{hhs} = D_2 + \lceil N_2/X \rceil * D_1 \qquad \text{(HHS1)}$$

where the first term is the cost of scanning C2 and the second term is the cost of scanning C1, and $\lceil N_2/X \rceil$ is the number of times C1 needs to be scanned.

The above cost formula assumes that all I/Os are sequential I/Os (i.e., both C1 and C2 are sequentially scanned in). This is reasonable only when each document collection is read by a dedicated drive with no or little interference from other I/O requests. If this is not the case, then some of the I/Os may become more costly random I/Os. We first consider the case when $N_2 \geq X$. The following interleaved I/O and CPU patterns can be observed. After each X documents in C2 are read in, for each document d in C1 read in, CPU will take some time to compute the similarities between the X documents and d. When the CPU is doing the computation, I/O resources may be allocated to other jobs. If this is the case, then the next document from C1 will use a random I/O, so does the read-in of the next X documents in C2. In other words, in the worst case, all documents in C1 will be read in using random I/O and for every X documents in C2, there will be a random I/O. The number of actual random I/Os for scanning documents in C1 once also depends on the document size and can be estimated as $\min\{D_1, N_1\}$ (if $S_1 \leq 1$, then D_1 should be used; otherwise, N_1 should be used). Therefore, when $N_2 \geq X$, in the worst scenario, the total I/O cost can be estimated as follows:

$$\text{hhr} = \text{hhs} + \lceil N_2/X \rceil * (1 + \min\{D_1, N_1\}) * (\alpha - 1)$$

When $N_2 < X$, then the entire collection C2 can be scanned in sequentially and held in the memory, and the remaining memory space $((X - N_2) * S_2)$ can be used to hold documents in C1. Therefore, C1 can be read in in $\lceil D_1/((X - N_2) * S_2) \rceil$ blocks and each block can be read in sequentially. In this case, we have

$$\text{hhr} = \text{hhs} + \lceil D_1/((X - N_2) * S_2) \rceil * (\alpha - 1)$$

5.2 Algorithm HVNL

Recall that a B+tree is maintained for each document collection for quickly locating the inverted file entry of any given term. The size of the B+tree can be estimated as follows. Typically, each cell in the B+tree occupies 9 bytes (3 for each term number, 4 for address and 2 for document frequency). If a document collection has N terms, then the size of the B+tree is approximately 9*N/P (only the leaf nodes are considered). The size is not terribly large. For example, for a document collection with 100,000 distinct terms, the B+tree takes about 220 pages of size 4KB. We assume that the entire B+tree will be read in the memory when the inverted file needs to be accessed and it incurs a one-time cost of reading in the B+tree.

Let X be the number of inverted file entries on C1 that can be held in the memory when the memory buffer is fully used. In addition to X inverted file entries, the memory (size B) also need to contain a document in C2 of size $\lceil S_2 \rceil$, a B+tree of size Bt_1, the non-zero similarities values between the document in C2 currently under processing and all documents in C1 and the list containing the terms whose corresponding inverted file entries are in the main memory (size $X|t\#|/P$). Therefore, X can be estimated as follows:

$$\text{X} = \text{floor}\left(\frac{B - \lceil S_2 \rceil - Bt_1 - 4 * N_1 \delta/P}{J_1 + |t\#|/P}\right)$$

If we assume that the read-in of the documents in C2 incurs sequential I/Os, then the I/O cost of HVNL can be estimated as follows:

$$\text{hvs} = \begin{cases} min\{D_2 + I_1 + Bt_1, \\ \quad D_2 + T_2 * q * \lceil J_1 \rceil * \alpha + Bt_1\}, & if\ X \geq T_1 \\ D_2 + T_2 * q * \lceil J_1 \rceil * \alpha + Bt_1, & if\ T_1 > X \geq T_2 * q \\ D_2 + X * \lceil J_1 \rceil * \alpha + Bt_1 + \\ \quad (N_2 - s - X1 + 1) * Y * \lceil J_1 \rceil * \alpha, & otherwise \end{cases}$$

where the first case corresponds to the case when X is greater than or equal to the total number of inverted file entries on C1 (i.e., T_1). In this case, we can either read in the entire inverted file on C1 in sequential order (this corresponds to the first expression in $min\{\}$) or read in all inverted file entries needed to process the query (the number is $T_2 * q$) in random order (this corresponds to the second expression in $min\{\}$. The memory is large enough to do this since $X \geq T_1 \geq T_2 * q$.); the second case corresponds to the case when the memory is not large enough to hold all inverted file entries on C1 but is large enough to hold all needed inverted file entries; the last expression is for the case when the memory is not large enough to hold all needed inverted file entries on C1. In this case, the second term is the cost of finding and reading in the inverted file entries on C1 which correspond to the terms in documents in C2 until the memory is fully occupied. Suppose the memory is just large enough to hold all the inverted file entries on C1 corresponding to the terms in the first $(s-1)$ documents in C2 and a fraction (X1) of the inverted file entries corresponding to the terms in the s-th document in C2 (i.e., the inverted file entries on C1 corresponding to the terms in the first $s + X1 - 1$ documents in C2 can be held in the memory). Let Y be the number of new inverted file entries that need to be read in when a new document in C2 is processed after the memory is fully occupied. Then the third term is the total cost of reading in new inverted file entries for processing the remaining documents in C2. We now discuss how s, X1 and Y can be estimated. First, the number of distinct terms in m documents in C2 can be estimated by $f(m) = T_2 - (1 - K_2/T_2)^m * T_2$. Therefore, s is the smallest m satisfying $q * f(m) > X$. Note that $(X - q * f(s-1))$ is the number of inverted file entries that can still be held in the memory after all the inverted file entries on C1 corresponding to the terms in the first $(s-1)$ documents in C2 have been read in and $(q * f(s) - q * f(s-1))$ is the number of new inverted file entries that need to be read in when the s-th document in C2 is processed, X1 can be estimated by $(X - q * f(s-1))/(q * f(s) - q * f(s-1))$. Finally, Y can be estimated by $(q * f(s + X1) - X)$.

As discussed in Section 5.1, it is possible that some or all of the I/Os of reading in the documents in C2 are random I/Os due to other obligations of the I/O device. If after inverted file entries are accommodated, there is still more memory space left, then the remaining memory space can be used to sequentially scan in multiple documents in C2 at a time. Based on this observation, when random I/Os are considered, the total I/O cost of HVNL can be estimated as:

$$\text{hvr} = \begin{cases} min\{D_2 + I_1 + Bt_1 + \lceil D_2/((X - T_1) * J_1) \rceil \\ \quad * (\alpha - 1), D_2 + T_2 * q * \lceil J_1 \rceil * \alpha + Bt_1 \\ \quad + \lceil D_2/((X - T_2 * q) * J_1) \rceil * (\alpha - 1)\}, & if\ X \geq T_1 \\ hvs + \lceil D_2/((X - T_2 * q) * J_1) \rceil \\ \quad * (\alpha - 1), & if\ T_1 > X \geq T_2 * q \\ hvs + \min\{D_2, N_2\} * (\alpha - 1), & otherwise \end{cases}$$

It would be easier to understand the above formula when compared with the formula for computing hvs. In the first expression in min{}, $(X - T_1) * J_1$ is the remaining memory space after all inverted file entries are accommodated.

5.3 Algorithm VVM

To avoid the much higher cost of random I/O, we can simply scan both inverted files on the two collections. During the parallel scan, if two inverted file entries correspond to the same term, then invoke the similarity accumulating process. Recall that we assumed that inverted file entries are stored in increasing order of the term numbers. Therefore, one scan of each inverted file is sufficient to compute all similarities if the memory is large enough to accommodate all intermediate similarities. Therefore, in this case, if all the I/Os are sequential I/Os, the total I/O cost of VVM is:

vvs = $I_1 + I_2$

Again, some or all of the I/Os could actually be random I/Os due to other obligations of the I/O device. In the worst case scenario, i.e., all I/Os are random I/Os, the total I/O cost of VVM can be estimated as:

vvr = $(\min\{I_1, T_1\} + \min\{I_2, T_2\}) * \alpha$

VVM usually requires a very large memory space to save the intermediate similarity values. If only nonzero similarities are stored, then the memory space for storing intermediate similarity values for VVM is $4\delta * N_1 * N_2/P$. When the memory space is not large enough to accommodate all intermediate similarity values, a simple extension to the algorithm VVM can be made (see Section 4.3). In this case, the total cost can be estimated by multiplying vvs (or vvr) by $\lceil SM/M \rceil$, where SM = $4\delta * N_1 * N_2/P$ is the total number of pages needed to store the intermediate similarities when all pairs of documents in the two collections are considered at the same time and M = $B - \lceil J_1 \rceil - \lceil J_2 \rceil$ is the available memory space for storing the intermediate similarities. Therefore, a more general formula for estimating the total I/O cost when all the I/Os are sequential I/Os can be given below:

vvs = $(I_1 + I_2) * \lceil SM/M \rceil$ (VVS)

and a more general formula for estimating the total I/O cost when all the I/Os are random I/Os is:

vvr = $(\min\{I_1, T_1\} + \min\{I_2, T_2\}) * \alpha * \lceil SM/M \rceil$

5.4 Comparisons

Algorithm HHNL uses two document collections as the input. It does not use any special data structures such as inverted files and B+trees. Thus, it is more easily applicable. It is also easier to implement. Since HHNL uses documents directly for similarity computation, it benefits quite naturally from any possible reductions to the number of documents in either or both collections resulted from the evaluation of selection conditions on non-textual attributes of the relevant relations. The memory space requirement of this algorithm for storing intermediate similarity values is generally small comparing with those of other algorithms.

Algorithm HVNL uses one document collection, one inverted file and the B+tree corresponding to the inner collection as the input. While the document collection is always scanned once, the access to inverted file entries is more complex. On the one hand, not all inverted file entries need to be read in. In fact, only those inverted file entries whose corresponding terms also appear in the other document collection need to be accessed. On the other hand, some inverted file entries may be read in many times due to their appearance in multiple documents in C2 although effort is made by the algorithm to reuse inverted file entries currently in the memory. It is expected that this algorithm can be very competitive in one of the following two situations: (1) One of the document collection, say C2, is much smaller than the other collection because in this case, it is possible that only a small fraction of all inverted file entries in the inverted file needs to be accessed. When C2 contains only one document, it becomes an extreme case of processing a single query against a document collection. Note that an originally large document collection may become small after conditions on attributes of the relevant relation are evaluated. (2) For the collection where documents are used, close documents in storage order share many terms and non-close documents share few terms. This increases the possibility of reuse of inverted file entries in the memory and reduces the possibility of re-read in of inverted file entries. This could happen when the documents in the collection are clustered.

Algorithm HVNL accesses inverted file entries in random order. As such, it has two negative effects on the I/O cost. One is that random I/Os are more expensive than sequential I/Os. The other is that even when an inverted file entry occupies a small fraction of a page, the whole page containing the entry has to be read in. Therefore, when the size of each inverted file entry is close to an integer, the competitiveness of HVNL will be increased. One thing bad about using the inverted file is that the size of the file remains the same even if the number of documents in the corresponding document collection can be reduced by a selection. The memory space requirement of HVNL for storing intermediate similarities is generally higher than that of HHNL but lower than that of VVM.

Algorithm VVM uses two inverted files as the input. As we discussed before, this algorithm has a very nice one-scan property, namely, it only needs to scan each inverted file once to compute the similarities regardless of the sizes of the two collections provided that the memory space is large enough to accommodate intermediate similarity values. Since the size of the inverted file on a collection is about the same as the size of the collection (when the collection is not reduced by other selections), VVM can be at least as efficient as HHNL as far as I/O cost is concerned. The major drawback of VVM is that it needs a very large space to save the intermediate similarities. Since the memory requirement for saving these similarities is proportional to $N_1 * N_2$, VVM is likely to perform well for document collections that are large in size but small in number of documents. Another disadvantage of VVM is that the sizes of the inverted files will remain the same even if the number of documents in the corresponding document collections can be reduced.

6. Simulation results

Due to the large number of parameters in the cost formulas of the algorithms presented, it is very difficult to compare the performance of these algorithms based on these formulas directly. In this section, different algorithms are compared based on simulation results. Our objective is to find out the impact of the variations of the parameters on the algorithms, in other words, we would like to find out in what situation an algorithm performs the best.

Three document collections which were collected by ARPA/NIST [5], namely, WSJ (Wall Street Journal), FR (Federal Register) and DOE (Department of Energy), are used in our simulation. The statistics of these collections are shown in the table below (the last three rows are estimated by us based on $|t\#| = 3$).

	WSJ	FR	DOE
#documents	98736	26207	226087
#terms per doc	329	1017	89
total # of distinct terms	156298	126258	186225
collection size in pages	40605	33315	25152
avg. size of a document	0.41	1.27	0.111
avg. size of an inv. fi. en.	0.26	0.264	0.135

Among the three document collections, FR has fewer but larger documents and DOE has more but smaller documents. The number of documents in WSJ lies between those of FR and DOE. So is the average size of documents in WSJ.

For all simulations, the page size P is fixed at 4KB, the fraction of the similarities that are non-zero δ is fixed at 0.1 and λ is fixed at 20 (note that only HHNL involves λ and it is not really sensitive to λ). The probability q is computed as follows:

$$q = \begin{cases} 0.8 * T_1/T_2, & if\ T_1 \leq T_2 \\ 0.8, & if\ T_2 < T_1 < 5 * T_2 \\ 1 - T_2/T_1, & if\ T_1 \geq 5 * T_2 \end{cases}$$

The formula says that, given the number of distinct terms in C2 (i.e., T_2), the smaller the number of distinct terms in C1, T_1, is, the smaller the probability that a term in C2 also appears in C1 will be; and when T_1 becomes much larger than T_2, then q will become closer to 1; otherwise, q is 0.8. Probability p can be computed in a similar manner.

For parameters B (memory size) and α, we assign a base value for each: B = 10,000 (pages) and α = 5. When the impact of a parameter is studied, we vary the values of the parameter while let the other parameter use its base value.

We conducted the following five groups of simulations.

Group 1: A real collection is used as both collection C1 and collection C2. Since there are three real collections (WSJ, FR and DOE) and two parameters (B and α), six simulations are conducted.

Group 2: Different real collections will be used as C1 and C2. B varies while α uses its base value. From the three real collections, six simulations are designed.

Group 3: While C1 and C2 continue to use real collections, only a small number of documents in C2 are used to participate the join. These experiments are used to investigate the impact of local selections. All simulations in this group use only the base values of the two parameters. Since there are three real collections, three simulation results are collected for this group.

Group 4: C1 again uses real collections but C2 uses collections with only a small number of documents. The difference between Group 3 and Group 4 is that the former uses a small number of documents (in C2) from an ORIGINALLY large collection C2 and the latter uses an ORIGINALLY small collection C2. This difference has the following impacts on the cost: (1) documents in C2 need to be read in randomly by the former but can be read in sequentially by the latter; and (2) the size of the inverted file and the size of the B+tree on collection C2 for the former are computed based on the original collection, not just the documents used. This will have an impact on the cost of VVM. In our experiments, after a real collection is chosen to be C1, C2 will be derived from C1. Again, all simulations in this group use only the base values of the two parameters and three simulations are conducted for this group.

Group 5: Both collection C1 and collection C2 use new collections but they remain to be identical. Each new collection is derived from a real collection by reducing the number of documents in the real collection and increasing the number of terms in each document in the real collection by the same factor such that the collection size remains unchanged. The simulations in this group are especially aimed at observing the behavior of Algorithm VVM. Again, only the base values of the two parameters are used and three simulation results are collected for this group.

Due to space limitation, the actual and detailed simulation results will not be presented here, and interested readers are referred to [11]. The summary of the simulation results are presented below.

6.1 Summary of simulation results

The following main points can be summarized from our extensive simulations.

1. The cost of one algorithm under one situation can differ drastically from that of another algorithm under the same situation. As a result, it is essential to choose an appropriate algorithm for a given situation.
2. If the number of documents in one of the two collections, say M, is originally very small or becomes very small after a selection, then HVNL has a very good chance to outperform other algorithms. Although how small for M to be small enough mainly depends on the number of terms in each document in the outer collection, M is likely to be limited by 100.

3. If the number of documents in each of the two collections is not very large (roughly $N_1 * N_2 < 10000 * B$) and both document collections are large such that none can be entirely held in the memory, then VVM (the sequential version) can outperform other algorithms.
4. For most other cases, the simple algorithm HHNL performs very well.
5. The costs of the random versions of these algorithms depict the worst case scenario when the I/O devices are busy satisfying different obligations at the same time. Except for VVM, these costs have no impact in ranking these algorithms.

Overall, the simulation results match well with our analysis in Section 5.4.

Since no one algorithm is definitely better than all other algorithms in all circumstances, it is desirable to construct an integrated algorithm that can automatically determine which algorithm to use given the statistics of the two collections $(N_1, N_2, K_1, K_2, T_1, T_2, p, q, \delta)$, system parameters (B, P, α) and query parameters (λ, selectivities of predicates on non-textual attributes). The sketch of an integrated algorithm can be found in [11].

7. Concluding remarks

In this paper, we presented and analyzed three algorithms for processing joins between attributes of textual type. From analysis and simulation, we identified, for each algorithm, the type of input document collections with which the algorithm is likely to perform well. More specifically, we found that HVNL can be very competitive when the number of documents in one of the two document collections is/becomes very small, and VVM can perform very well when the number of documents in each of the two collections is not very large and both document collections are large such that none can be entirely held in the memory. In other cases, HHNL is likely to be the top performer. Since no one algorithm is definitely better than all other algorithms, we proposed the idea of constructing an integrated algorithm consisting of the basic algorithms such that a particular basic algorithm is invoked if it has the lowest estimated cost. We also indicated that the standardization of term numbers will be very useful in multidatabase environments.

Further studies in this area include (1) investigate the impact of the availability of clusters on the performance of each algorithm; (2) develop cost formulas that include CPU cost and communication cost; (3) develop algorithms that process textual joins in parallel; and (4) more detailed simulation and experiment.

References

[1] U. Dayal and H-Y Hwang, *View Definition and Generalization for Database Integration in a Multidatabase system.* IEEE TSE, 1984.

[2] W. Du, R. Krishnamurthy, and M. C. Shan, *Query Optimization in Heterogeneous Databases.* VLDB Conference, 1992.

[3] S. Dumais and J. Nielson, *Automating the Assignment of Submitted Manuscripts to Reviewers.* ACM SIGIR, 1992.

[4] S. Ghose, *File Organization: The Consecutive Retrieval Property.* Comm. of ACM, 1972.

[5] D. Harman, *Overview of the first text retrieval conference* edited by D. Harman, Computer Systems Technology, U.S. Department of Commerce, NIST, 1993.

[6] W. Litwin, L. Mark, N. Roussopoulis, *Interoperability of Multiple Autonomous Databases.* ACM Computing Surveys, September 1990.

[7] H. Lu, B. Ooi and C. Goh, *On Global Multidatabase Query Optimization.* SIGMOD Record, December 1992.

[8] L. Lilian, and B. Bhargava, *A Scheme for Batch Verification of Integrity Assertions in a Database System.* IEEE TSE, Nov. 1984.

[9] W. Meng, C. Yu, W. Kim, G. Wang, T. Pham and S. Dao, *Construction of Relational Front-end for Object-Oriented Database Systems.* The 9-th IEEE Conference on Data Engineering, April 1993.

[10] W. Meng, A. Kamada, Y-H. Chang, *Transformation of Relational Schemas to Object-Oriented Schemas.* Compsac'95, August 1995.

[11] W. Meng, C. Yu, W. Wang, N. Rishe, *Performance Analysis of Several Algorithms for Processing Joins between Textual Attributes. CS-TR-95-07*, Department of CS, SUNY at Binghamton, 1995.

[12] G. Salton and M. McGill, *Introduction to Modern Information Retrieval.* McGraw-Hill, 1983.

[13] A. Sheth, and J. Larson, *Federated Database Systems for Managing Distributed, Heterogeneous, and Autonomous Databases.* ACM Computing Surveys, September 1990.

[14] R. Swaminathan, and D. Wagner, *On the Consecutive-Retrieval Problem.* SIAM J. Comput. April 1994.

[15] C. Yu, Y. Zhang, W. Meng, W. Kim, G. Wang, T. Pham, and S. Dao: Translation of Object-Oriented Queries to Relational Queries. *11-th IEEE Conf. on Data Engineering*, March 1995.