PERFORMANCE ANALYSIS OF THREE TEXT-JOIN ALGORITHMS

Weiyi Meng[1], Clement Yu[2], Wei Wang[3], Naphtali Rishe[4]

Abstract

When a multidatabase system contains textual database systems (i.e., information retrieval systems), queries against the global schema of the multidatabase system may contain a new type of joins — joins between attributes of textual type. Three algorithms for processing such type of joins are presented and their I/O costs are analyzed in this paper. Since such type of 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. An integrated algorithm that automatically selects the best algorithm to use is also proposed.

Index Terms: Query processing, textual database, information retrieval, join algorithm, multidatabase.

1 Introduction

Researches in *multidatabase system* have been intensified in recent years [4, 5, 9, 13, 12, 16, 19]. 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).

The global schema of a multidatabase system, integrated from local database schemas, provides an overall picture of all sharable data in the local systems. The global query language can be used to specify queries against the global schema, which will be referred to as *global queries* thereafter, and to retrieve data represented by the global schema. For example, if the global schema is in relational data model, then SQL can be used as the global query language. Since the multidatabase system considered in this paper contains

[1]Department of Computer Science, State University of New York at Binghamton, Binghamton, NY 13902-6000. Email: meng@cs.binghamton.edu.

[2]Department of Electrical Engineering and Computer Science, University of Illinois at Chicago, Chicago, IL 60607. Email: yu@eecs.uic.edu.

[3]Computer Science Department, UCLA, Los Angeles, CA 90024. Email: weiwang@cs.ucla.edu.

[4]School of Computer Science, Florida International University, Miami, FL 33199. Email: rishen@fiu.edu.

IR components and relational components, the global query language must be capable of accommodating both structured data and unstructured data. An SQL-based query language that can serve such a purpose has been proposed in [1]. In this paper, we extend the features of this language to specify our queries.

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. Although other types of comparators between textual attributes may exist, the SIMILAR_TO operator is a key operator for textual data and therefore we concentrate on this operator in this paper.

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 [6], 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 is small, efficient processing strategy of the join is not their concern. Instead, the emphasis of that work is on the accuracy of the automated match. A somewhat related problem is the *consecutive retrieval problem* [7, 17] 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 with 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 of 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 need to be processed 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 given above. As we will see in this paper, the availability of inverted files means the applicability of certain algorithms. The clustering problem in IR systems [14] 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 as described here when the two document collections involving the join are identical.

A straightforward way exists for processing joins between textual attributes in a multidatabase environment. This method can be described as follows: Treat each document in one collection as a query and process each such query against the other collection independently to find the most similar documents. However, this method is extremely expensive since either all documents in one of the two collections are searched or the inverted file of that collection is utilized once for processing each document in the other collection. As an example, consider the *Smart* system [3] developed at Cornell University. The *Smart* system uses inverted file to process user queries. If the collection whose documents are used as queries has a large number of documents, then using the inverted file of the other collection to process each query independently can easily incur a cost which is several orders of magnitude higher than that of a better join algorithm (see Section 6). Therefore, it is very important to develop efficient algorithms for processing joins between textual attributes. This paper has the following contributions: (1) We present and analyze three algorithms for processing joins between attributes of textual type. (2) Cost functions based on the I/O cost 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 different 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. An integrated algorithm that automatically selects the best algorithm to use is also included in this section. 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 relation Applicants contains information of applicants for job positions in relation Positions, and 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. All job descriptions will be listed as output by the former. However, a job description may not be listed in the output by the latter if it is not among the λ most similar job descriptions to any resume. Later, we will see that the asymmetry of the operator SIMILAR_TO has some impact on the evaluation strategy.

There are some important differences between joins in relational database systems and the join between two textual attributes. Consider the relational join R1.A θ R2.A, where θ is a comparator such as = and >. Given a tuple t1 of R1 and a tuple t2 of R2, if t1[A] θ t2[A] is true, then we immediately know that t1 and t2 satisfy the join. However, for a given resume r and a given job description j, there is no way for us to know immediately whether or not r SIMILAR_TO(λ) j is true since 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 contain textual attributes, it is possible that only a subset of the set of 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". The impact of selections will also be addressed.

3 Assumptions and Notations

Using the vector representation [14], 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 ascending term numbers. 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. Another method is to establish a mapping between the corresponding numbers identifying the same term. Such a mapping structure, usually a table with two columns, if not stored in the main memory, can substantially degrade the performance. Assuming $|t\#| = 3$, then approximately 150 pages, each of size 4KB, are needed for the mapping structure to accommodate 100,000 distinct terms. Since the total size of the mapping structure is less than 1MB, it is likely that the mapping structure can be held in the memory. 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 [14], 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 ascending document numbers. 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, i.e, no space is left unused in each page except possibly the last page)

p — the probability that a term in collection C1 also appears in collection C2 q — the probability that a term in collection C2 also appears in collection C1

α — the cost ratio of a random I/O over a sequential I/O

P — page size in bytes (4KB)

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 $N_i * \lceil S_i \rceil$ and the total cost will approximately be $N_i * \lceil S_i \rceil * \alpha$, where $\lceil X \rceil$ denotes the ceiling of X and α is the cost ratio of a random I/O over a sequential I/O due to the additional seek and rotational delay of a random read. Similarly, we assume that inverted file entries on each collection are stored in consecutive storage locations in ascending 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, then 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. Cost analysis and comparisons for a distributed environment will be conducted in the future.

4 Algorithms

In this section, we present three algorithms for processing joins on textual attributes. These algorithms will be analyzed and compared in the next two sections. We assume 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 generally be read in sequentially resulting in 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. We consider the forward order first.

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 which will be discussed later. 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 of the memory need to be reserved for C1. We also need to reserve the space to save the 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 = $(B - \lceil S_1 \rceil)/(S_2 + 4\lambda/P)$, where P is the size of a page in bytes.

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 in C2 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 similarities 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;
               }
           }
    }
```

If λ is large, then a heap structure can be used to find the smallest of the λ largest similarities in the above algorithm.

We now consider the backward order. When C1 is used as the outer collection to evaluate the join, C2 will be scanned for each set of documents in C1 currently in the memory. Let $d1$ be the first document in C1 read in the memory. After C2 is scanned, the N_2 similarities between $d1$ and every document in C2 are computed. Since for each document in C2, we need to find the λ documents in C1 that are most similar to it, we need to keep track of the λ documents in C1 that have the largest λ similarities for each document in C2. This means that we need to keep track of the $\lambda * N_2$ similarities during the backward order evaluation. In other words, we need a memory space of size $4\lambda * N_2/P$ to keep these similarities. Compared with the forward order which requires $4\lambda * X1/P$ pages to keep track of the needed similarities, more memory space is needed to save the similarities for the backward order. This will have an adverse impact on the performance of the backward order. As a result, the forward order is likely to perform better than the backward order when the two document collections have about the same size. However, when C1 is much smaller than C2, then the backward order can still outperform the forward order. For example, if C1 can be entirely held in the memory, then only one scan of each collection is needed to process the join with the backward order no matter how large C2 is.

4.2 Algorithm HVNL

This algorithm uses the documents in one collection and the inverted file for the other collection to compute the similarities. In an information retrieval 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 non-zero entries in the document-term matrix be accessed. A more efficient way is to use the inverted file on the document collection to process the query. This method is used in the *Smart* system [3]. The advantage of this method is that it only needs to access those non-zero entries in the columns of the document-term matrix which correspond 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. Algorithm 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, $0 < \delta < 1$.

A straightforward way to process the join is to go through the above process for each document in C2 independently. That is, read in each document d in C2 in turn and while d is in the memory, read in all inverted file entries on C1 corresponding to terms in d to process d (note that not all terms in d will necessarily appear in C1). The problem with this straightforward method is its lack of coordination between the processing of different documents in C2. As a result, if a term appears in K documents in C2, then the inverted file entry of the term (assume that it also appears in C1) on C1 will be read in K times. Algorithm HVNL is designed to re-use the inverted file entries that are read in the memory for processing earlier documents to process later documents to save I/O cost. Due to space limitation, usually not all inverted file entries read in earlier can be kept in the memory. Therefore, the algorithm also needs a policy for replacing an inverted file entry in the memory by a new inverted file entry. 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. This means that each newly read in document will be scanned twice in the memory. The first scan is to find the terms whose corresponding inverted file entries are already in the memory and the second scan is to process other terms. A list that contains the terms whose corresponding inverted file entries are in the memory will be maintained. Note that when not all inverted file entries that are read in earlier can be kept in the memory, it is still possible to read in an inverted file entry more than one time. Note also that the worst case scenario for algorithm HVNL is that for each document in C2 under consideration, none of its corresponding inverted file entries is currently in the memory. In this case, algorithm HVNL deteriorates into the straightforward method.

We now present the algorithm HVNL:

For each document d in C2

    {For each term t in d

        If t also appears in C1

            If the inverted file entry of t on C1 (I_1^t) is in the memory

                accumulate similarities;

For each term t in d

If t also appears in C1

{If the inverted file entry of t on C1 (I_1^t) is not in the memory

If the available memory space can accommodate I_1^t

read in I_1^t;

Else

find the inverted file entry in the memory with the lowest document frequency and replace it with I_1^t;

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 process, then the likelihood of an inverted file entry already in the memory to be reused can be increased. For example, consider three documents each with three terms: D1 = {t1, t2, t3}, D2 = {t2, t3, t4} and D3 = {t3, t4, t5}. Suppose terms with smaller subscripts have lower document frequencies. Suppose the memory buffer is only large enough to hold three inverted file entries. If D1, D2 and D3 are processed in the given order, then each inverted file entry needs to be read in exactly once. However, if the processing order is D1, D3 and D2, then the inverted file entry corresponding to t2 will be read in twice and all other inverted file entries will be read in exactly once. Clearly, for this example, order {D1, D2, D3} is better than order {D1, D3, D2}.

An order is *optimal* if it incurs the minimum I/O cost. The question is can an optimal order be found efficiently. Unfortunately, as shown by the proposition below, the problem of finding an optimal order is NP-hard.

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

Proof: It was shown in [11] 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. It can be seen that the optimal order problem in our case is essentially the same as the optimal order problem in OBIAV because the following correspondences between the two problems can be easily established: processing a document in C2 corresponds to verifying an integrity constraint; the need to read in a set of inverted file entries for processing each document in C2 corresponds to the need to bring in a set of pages for verifying each integrity constraint; that an inverted file entry read in for processing one document may be used for processing another document corresponds to that a page brought in for verifying one integrity constraint may be used for verifying another integrity constraint. Therefore, the optimal order problem in our case is also NP-hard. ■

We decide not to pursue the issue of finding an optimal order further because in addition to its NP-hard nature, there is another problem associated with any optimal order, that is, by reading in documents in any order rather than their storage order, more expensive random I/Os will be incurred.

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, algorithm VVM can be at least as good as algorithm HHNL because algorithm 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. Algorithm 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 $N_1 * N_2$ (independent of the number of terms in each document), which can be so large such that algorithm VVM can not be run at all. In summary, algorithm 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. Another situation that algorithm VVM may do well is when the vocabularies of the two document collections are very different. For example, one collection is on medicine and the other is on computer science. In this case, the number of non-zero similarities between documents in the two collections is likely to be small.

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 ascending term numbers. Therefore, one scan of each inverted file is sufficient (very much like the merge phase of merge sort). 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

In this section, we provide analysis of the I/O cost of each algorithm presented in Section 4.

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:

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

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, the 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:

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

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

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

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 needs 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:

$$X = \lfloor \frac{B - \lceil S_2 \rceil - Bt_1 - 4 * N_1 \delta / P}{J_1 + |t\#|/P} \rfloor$$

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:

$$hvs = \begin{cases} min\{D_2 + I_1 + Bt_1, D_2 + T_2 * q * \lceil J_1 \rceil * \alpha + Bt_1\} & if\ X \geq T_1 \\ D_2 + T_2 * q * \lceil J_1 \rceil * \alpha + Bt_1, & if\ T_1 > X \geq T_2 * q \\ D_2 + X * \lceil J_1 \rceil * \alpha + (N_2 - s - X1 + 1) * Y * \lceil J_1 \rceil * \alpha + Bt_1, & otherwise \end{cases} \quad (HVS1)$$

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 of the necessary 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:

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

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.

With slight modification on similarity accumulation, C1 can be used as the outer collection to process the query. In this case, the memory space needed to store intermediate similarities will be $4\lambda\delta N_2/P$. The cost of the backward order can be estimated in the same way as in the case of the forward order.

5.3 Algorithm VVM

To avoid the much higher cost of random I/O's, 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 the inverted file entries are stored in ascending 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, if all the I/Os are sequential I/Os, the total I/O cost of the algorithm 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 the algorithm VVM can be estimated as:

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

Algorithm VVM usually requires a very large memory space to save the intermediate similarity values. If only non-zero similarities are stored, then the memory space for storing intermediate similarity values for the algorithm 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. Each of the two document collections needs to be scanned at least once, which constitutes the lower bound of the I/O cost of this algorithm. Algorithm HHNL does not use any special data structures such as inverted files and B+trees. Thus, it is more easily applicable and easier to implement. Since algorithm HHNL uses documents directly for similarity computation, it benefits quite naturally from any possible reductions to the number of documents in either one 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 compared 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 appearances 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. In this case, it is likely that only a small fraction of all inverted file entries in the inverted file needs to be accessed. This means that only a small portion of the document-term matrix corresponding to C1 will be accessed in this case. In contrast, if algorithm HHNL is used, then the entire matrix needs to be accessed at least once even when C2 can be held entirely in the memory.

   When C2 contains only one document, this situation becomes an extreme case of processing a single query against a document collection. As we have mentioned before, using the inverted file to process a single query has been shown in IR to be superior to using documents directly. 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 reusing inverted file entries in the memory and reduces the possibility of re-reading in 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. In other words, if e is the size of an inverted file entry, we need to read in $\lceil e \rceil$ even if e is very small, say 0.1. Therefore, when the size of each inverted file entry is close to an integer, the competitiveness of algorithm HVNL will be increased. Algorithm HVNL uses primarily two data structures, one is the inverted file and the other is the B+tree for the terms. One disadvantage of 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 unless we construct another inverted file for the reduced set, which is highly unlikely due to the cost involved. The memory space requirement of algorithm HVNL for storing intermediate similarities is higher than that of algorithm HHNL but lower than that of algorithm 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. When the memory space is large enough to accommodate intermediate similarity values, algorithm VVM can be at least as efficient as algorithm HHNL as far as I/O cost is concerned. The major drawback of algorithm VVM is that it needs a very large memory space to save the intermediate similarities. There are two situations that algorithm VVM is likely to perform well. The first is when the document collections are large in size but small in number of documents. The second is when the vocabularies of the two document collections are very different. In both of the two situations, the number of non-zero similarities between documents in the two collections is likely to be small. Another disadvantage of algorithm 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, the algorithms are compared based on simulation results computed from the cost formulas derived in Section 5. Our objective is to identify 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.

The statistics of three document collections which were collected by ARPA/NIST [8], 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 Table 1 (the last three rows are estimated by us based on $|t\#| = 3$).

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 algorithm HHNL and the backward order of algorithm HVNL involve λ and none is really sensitive to λ if it is not very large, say in the hundreds). The probability q is computed as follows:

	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

Table 1: Statistical information of several document collections

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

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 $\alpha = 5$. 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 present the following five groups of simulation results.

Group 1: In this group, a real collection will be 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 simulation results will be collected.

Group 2: In this group, different real collections will be used as C1 and C2. B will vary while α will use its base value. From the three real collections, six simulations can be designed.

Group 3: In this group, while C1 and C2 will continue to use real collections, only a small number of documents in C2 will be used to participate in 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 will be collected in this group.

Group 4: In this group, C1 again will continue to use real collections, but C2 will be 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 still 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 algorithm 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. Since there are three real collections, three simulation results will be collected in this group.

Group 5: In this group, both collection C1 and collection C2 will use new collections but they will 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 to be the same. The simulations in this group are especially aimed at observing the behavior of algorithm VVM. Again, only the base values of the two parameters will be used and three simulation results will be collected in this group since there are three real collections.

For space consideration, the simulation results for the backward order approach will not be presented. Notice that the backward order approach makes a difference only when HHNL and HVNL are used (see the discussions in Section 4.1 and Section 5.2). Compared with the forward order, the backward order requires more memory space to store intermediate similarities. As a result, the backward order with outer collection A1 and inner collection A2 incurs a somewhat higher cost than the forward order with outer collection B1 and inner collection B2 when A1 and B1 are the same collection and A2 and B2 are the same collection.

For all the figures in this section, a value k on y-axis is equivalent to 10^k sequential page I/Os. For figures 1, 3 and 4, each unit on x-axis is equivalent to 10,000 pages.

Simulation results in Group 1

The following simulations are conducted in this group:

Simulation 1: C1 = C2 = WSJ, $\alpha = 5$, B changes from 10000 to 50000 with an increment of 5000

Simulation 2: C1 = C2 = FR, $\alpha = 5$, B changes from 10000 to 50000 with an increment of 5000

Simulation 3: C1 = C2 = DOE, $\alpha = 5$, B changes from 10000 to 50000 with an increment of 5000

Simulation 4: C1 = C2 = WSJ, B = 10000, α changes from 3 to 10 with an increment of 1

Simulation 5: C1 = C2 = FR, B = 10000, α changes from 3 to 10 with an increment of 1

Simulation 6: C1 = C2 = DOE, B = 10000, α changes from 3 to 10 with an increment of 1

The following observations can be made from the result of simulation 1 (see Figure 1).

1. Algorithm HHNL outperforms the other two algorithms, especially when B is small.

2. There are several reasons that algorithm HVNL performs poorly. First, the outer document collection has too many documents ($N_2 = 98,736$) which causes repeated read-ins of many inverted file entries on C1. Second, algorithm HVNL requires more random I/Os. Third, both S_2 and J_1 are not close to integers and as a result, for each document or inverted file entry read in, algorithm HVNL incurs more than twice as much cost as that by algorithm HHNL (1 versus 0.41 for document and 1 versus 0.26 for inverted file entry).

3. The main reason that algorithm VVM performs very poorly is because the memory requirement for storing intermediate similarities (952,031 pages) is much greater than the available memory. As a result, many scans of the two inverted files are needed to process the join.

4. All algorithms perform better with larger available memory. When B = 45,000 or larger, one document collection or an inverted file can be held in the memory in its entirety. When this happens, algorithm HHNL and algorithm HVNL have very similar performances since in this case, algorithm HHNL scans each of the two document collections once and algorithm HVNL scans one document collection and one inverted file which has the same size as a document collection.

Similar observations as made from the result of simulation 1 can also be made from the results of simulation 2 and simulation 3 (not shown). Relatively speaking, the performance of algorithm VVM in simulation 2 has the largest improvement due to the larger size of documents and fewer number of documents. However, the memory requirement for storing intermediate similarities in this case (67,071 pages) is still too large for the available memory to handle and at least two scans of the two inverted files are used to process the join. Not surprisingly, the relative performance of algorithm VVM in simulation 3 has become much worse due to the smaller size of documents and larger number of documents.

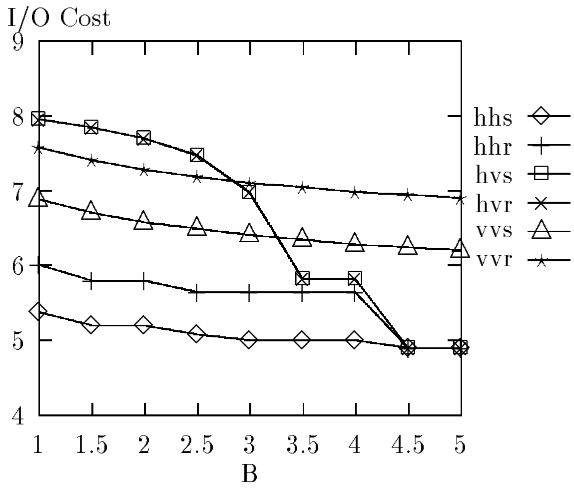

Figure 1: Result of Simulation 1

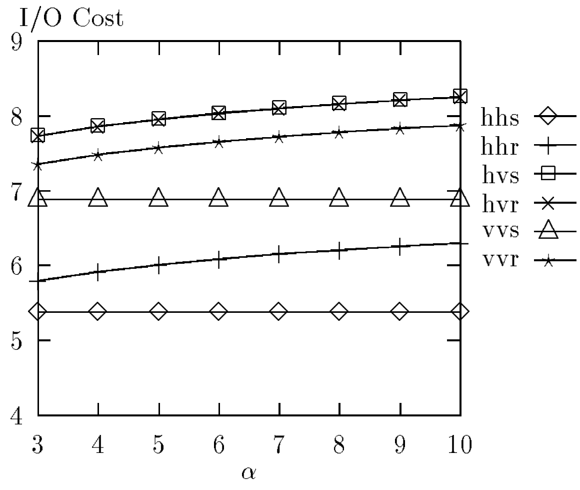

Figure 2: Result of Simulation 4

The following observations can be made from the result of simulation 4.

1. Algorithm HHNL is the best performer among the three algorithms.

2. hhs and vvs are independent of α because they involve no random I/Os.

3. Others become worse when α increases.

4. Algorithm HVNL is more sensitive to larger α.

Similar observations can be made from the results of simulation 5 and 6.

Simulation results in Group 2

In this group, different real collections will be used as C1 and C2 and the base values will be used for B and α. From the three real collections, the following six simulations can be designed.

Simulation 7: C1 = WSJ, C2 = FR, $\alpha = 5$*, B changes from 10000 to 50000 with an increment of 5000*

Simulation 8: C1 = FR, C2 = WSJ, $\alpha = 5$*, B changes from 10000 to 50000 with an increment of 5000*

Simulation 9: C1 = FR, C2 = DOE, $\alpha = 5$*, B changes from 10000 to 50000 with an increment of 5000*

Simulation 10: C1 = DOE, C2 = FR, $\alpha = 5$*, B changes from 10000 to 50000 with an increment of 5000*

Simulation 11: C1 = WSJ, C2 = DOE, $\alpha = 5$*, B changes from 10000 to 50000 with an increment of 5000*

Simulation 12: C1 = DOE, C2 = WSJ, $\alpha = 5$*, B changes from 10000 to 50000 with an increment of 5000*

Comparing the result of simulation 7 with the result of simulation 8 (see Figures 3 and 4), the following observations can be made.

1. While algorithm HHNL is the best performer in simulation 7, algorithm HVNL sometimes beats HHNL in simulation 8. The reason is that while algorithm HHNL lets the outer collection use as much memory

space as possible, algorithm HVNL lets the inner collection use as much memory space as possible. For example, consider Figure 4 when B = 35,000 (i.e., B = 3.5 in the figure). In this case, the entire inverted file on FR can be held in the memory. As a result, when algorithm HVNL is used, only one scan of WSJ and the inverted file on FR is needed to process the join. However, when algorithm HHNL is used, the memory is not large enough to hold the entire outer collection WSJ. As a result, one scan of WSJ and two scans of the inverted file on FR are needed to process the join when algorithm HHNL is used.

2. There is no change on the cost of algorithm VVM because it is completely symmetric to the two document collections.

3. When none of the two collections can be entirely held in the memory, we get mixed results for algorithm HHNL, that is, sometimes, it has better result in simulation 7 than that in simulation 8, but sometimes the opposite is true. When only the smaller collection can be held in the memory, better performance can be achieved using the smaller collection as the outer collection. This is the reason that algorithm HHNL has better result in simulation 7 than that in simulation 8 when B becomes 35,000 or larger. This observation also supports our earlier argument in Section 4.1 that the backward order can outperform the forward order if the backward order implies a much smaller outer collection.

4. The situation for algorithm HHNL is reversed for algorithm HVNL. The reason is that while algorithm HHNL lets the outer collection use as much memory space as possible, algorithm HVNL lets the inner collection use as much memory space as possible.

Similar observations made above between the result of simulation 7 and the result of simulation 8 can also be made between the result of simulation 9 and the result of simulation 10, as well as between the result of simulation 11 and the result of simulation 12 (the results of simulations 9 — 12 are not shown).

Simulation results in Group 3

In this group, C1 and C2 will continue to be real collections but only a small number of documents in C2 will be used to participate the join. Let M be the number of such documents in C2. Since $M << N_2$, we should read each of the M documents individually in random order. As a result, the cost of reading in the M documents will be $M * \lceil S_2 \rceil * \alpha$. Based on this, we have the following new formula for hhs:

$$hhs = M * \lceil S_2 \rceil * \alpha + \lceil (M/X) * D1 \rceil \qquad (HHS2)$$

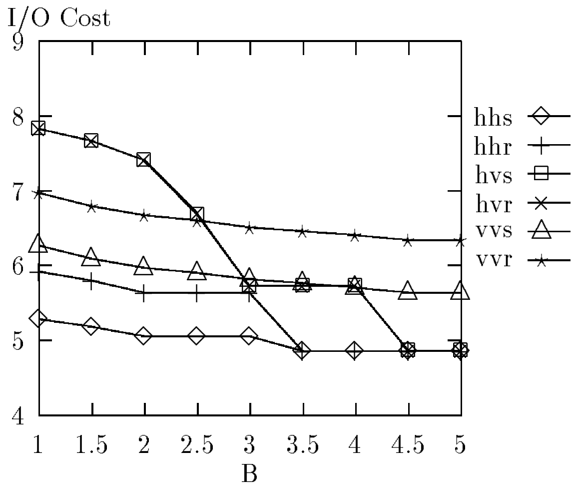

Figure 3: Result of Simulation 7

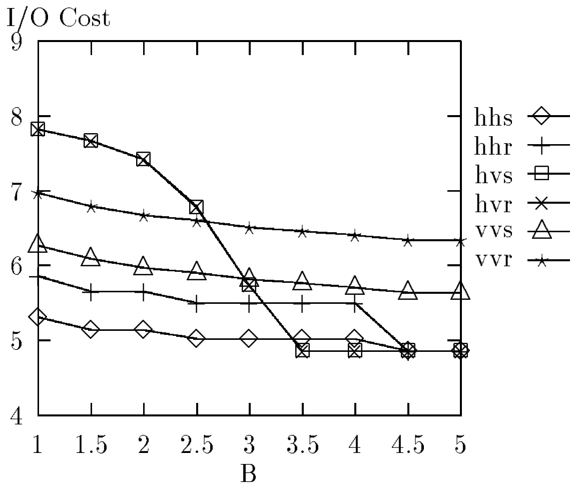

Figure 4: Result of Simulation 8

Since M is small, it is likely that all of the M documents in C2 can be held in the memory. In addition, the remaining memory space ($(X - M) * S_2$) can be used to read in as many documents in C1 as possible. As a result, we have the following formula for *hhr*:

$$hhr = hhs + \lceil D_1/((X - M) * S_2)\rceil * (\alpha - 1)$$

To compute *hvs* and *hvr*, we need to estimate the number of distinct terms in the M documents. This number can be estimated by $f(M) = T_2 - (1 - K_2/T_2)^M * T_2$. The cost formula for *hvs* is the same as that in Section 5.2 except D_2 is replaced by $M * \lceil S_2 \rceil * \alpha$ and T_2 is replaced by $f(M)$. Let this new formula be denoted by (**HVS2**). Since all I/O's in *hvs* have become random I/Os, $hvr = hvs$.

The cost formulas for *vvs* and *vvr* remain the same. However, the memory requirement for storing the intermediate similarities is now reduced to $4 * N_1 * M * delta/P$. Other quantities such as the size of inverted file entries and the size of the B+tree on collection C2 remain as before.

The following three simulations are carried out:

Simulation 13: C1 = C2 = WSJ, B = 10000, α = 5, M changes from 5 to 50 with an increment of 5

Simulation 14: C1 = C2 = FR, B = 10000, α = 5, M changes from 5 to 50 with an increment of 5

Simulation 15: C1 = C2 = DOE, B = 10000, α = 5, M changes from 5 to 50 with an increment of 5

The following observations can be obtained from the result of simulation 13 (see Figure 5).

1. When M is very small ($\leq$ 30), algorithm HVNL outperforms others as expected. Algorithm HHNL becomes the best performer when M becomes larger.

2. Since M is so small, the M documents can easily fit into the memory. As a result, algorithm HHNL

requires only one scan of the inner document collection in addition to reading in the M documents from the outer collection.

3. In this case, the memory is able to accommodate all intermediate similarities for algorithm VVM. The reason that algorithm VVM incurs much higher cost than algorithm HHNL is because the size of the inverted file on collection C2 did not change although only a small number of documents in C2 are used.

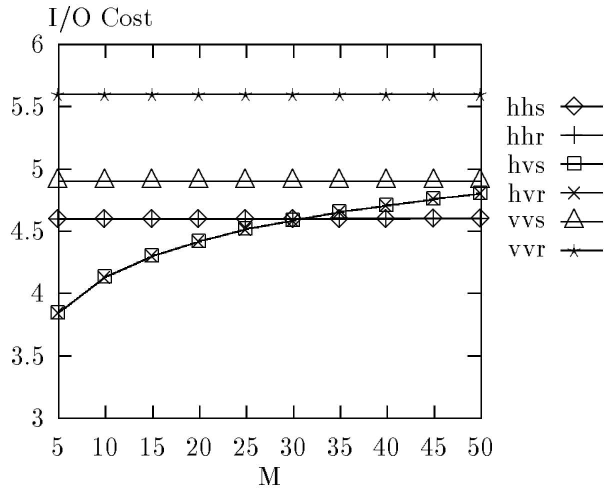

Figure 5: Result of Simulation 13

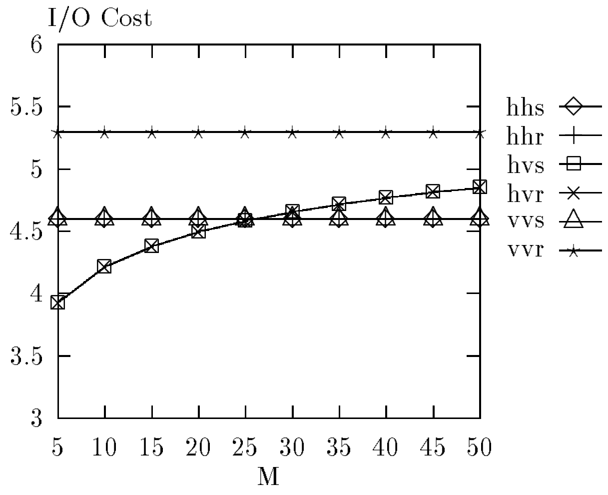

Figure 6: Result of Simulation 16

Comparing the result of simulation 14 (not shown) with the result of simulation 13, a noticeable difference is that the relative performance of algorithm HVNL deteriorated — algorithm HVNL becomes worse than algorithm HHNL before M reaches 10. This is because each document in FR contains much more terms than each document in WSJ and therefore more inverted file entries need to be read in by algorithm HVNL for processing a document in FR.

Comparing the result of simulation 15 (not shown) with the result of simulation 13, a noticeable difference is that the relative performance of algorithm HVNL is improved — algorithm HVNL outperforms algorithm HHNL even after M reaches 50. This is because each document in DOE contains much fewer terms than each document in WSJ and therefore fewer inverted file entries need to be read in by algorithm HVNL for processing a document in DOE.

For space consideration, we do not present simulation results for situations when the numbers of documents in both collections are reduced by selections. However, it is not difficult to see that comparing with the situation when only one collection is reduced, algorithm HHNL will benefit the most when both collections are reduced.

Simulation results in Group 4

In this group, C1 will continue to use real collections, but C2 will be collections with only a small number of documents. Since we do not have real collections that contain a small number of documents, we derive such a collection from a real collection. This turns out to be quite easy. From a given document collection, we first keep its document size and then decide the number of documents we want in the new collection. From this number, say M, the number of distinct terms in the new collection can be computed by $f(M)$. Now, all key statistics of the new collection become available. With these statistics, the cost formulas in Section 5 can be used to find the cost of each algorithm.

The following three simulations are conducted in the group:

Simulation 16: C1 = C2 = WSJ, B = 10000, α = 5, M changes from 5 to 50 with an increment of 5

Simulation 17: C1 = C2 = FR, B = 10000, α = 5, M changes from 5 to 50 with an increment of 5

Simulation 18: C1 = C2 = DOE, B = 10000, α = 5, M changes from 5 to 40 with an increment of 5

Comparing the result of simulation 16 (see Figure 6) with that of simulation 13 (see Figure 5), the following observations can be made.

1. There is little change for algorithm HHNL. Since M is so small, reading in the M documents sequentially or randomly makes little difference.

2. Algorithm HVNL degraded somewhat. This is the effect of q — the probability that a term in collection C2 also appears in collection C1. In Simulation 13, q is computed based on the original T_1 and T_2. Since $T_1 = T_2$, $q = 0.8$ is computed. In Simulation 16, q is computed based on the original T_1 and the new $f(M)$. Since $f(M)$ is much smaller than T_1, q between 0.92 to 0.99 are computed using the formula. Higher q values imply more inverted file entries on collection C1 need to be read in and as a result, the performance of algorithm HVNL is down.

3. The cost of algorithm VVM is reduced substantially. The main reason behind the reduction is the reduction of the size of the inverted file on C2. In Simulation 13, the size is computed based on the original C2, but in Simulation 16, the size is computed based on the reduced collection.

Similar observations as above can be made for simulation 17 and simulation 18.

Simulation results in Group 5

In this group, both C1 and C2 will use new collections but they will remain to be identical. Each new collection is derived from a real collection by reducing the number of documents and increasing the number of terms in each document in the real collection by the same factor F to ensure that the collection size remains to be the same.

The following three simulations are carried out:

Simulation 19: C1 = C2 are derived from WSJ, B = 10000, α = 5, the decreasing (increasing) factor changes from 1 to 13 with an increment of 2

Simulation 20: C1 = C2 are derived from FR, B = 10000, α = 5, the decreasing (increasing) factor changes from 1 to 5 with an increment of 1

Simulation 21: C1 = C2 are derived from DOE, B = 10000, α = 5, the decreasing (increasing) factor changes from 1 to 28 with an increment of 3

The following observations can be made from result of simulation 19 (see Figure 7).

1. When factor F is small (≤ 5), algorithm HHNL outperforms other algorithms. However, when F is 7 or larger, the sequential version of algorithm VVM (i.e., *vvs*) becomes the best performer.

2. *vvs* decreases rapidly as F increases as expected. When F reaches 11, all intermediate similarities can be held in the memory. As a result, *vvs* reaches its lower bound — each inverted file is scanned once. When F = 11, the number of documents in the collection is reduced to 8,976 and the number of terms in each document becomes 3,619.

3. *hvs* and *hvr* are insensitive to the changes.

4. *hhr* decreases as F increases. This is because as F increases, the number of documents in C1 decreases. Since the number of random I/Os is bounded by the the number of documents in C1, *hhr* decreases as a result.

Similar observations as for Simulation 19 can be made for Simulation 20 with the only difference that *vvs* reaches its minimum faster for the latter. The reason is that the number of documents in FR is originally much smaller than that in WSJ. Again, similar observations as for Simulation 19 can be made for Simulation 21 with the only difference that *vvs* reaches its minimum slower for the latter. The reason is that the number of documents in DOE is originally much larger than that in WSJ.

6.1 Summary of the Simulation Results

The following main points can be summarized from the above extensive simulations.

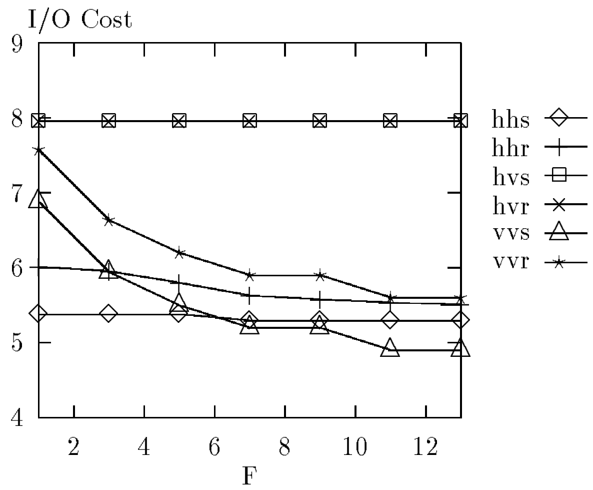

Figure 7: Result of Simulation 19

1. The cost of one algorithm under one situation can differ drastically from that of another algorithm under the same situation. For example, in Simulation 1, Algorithm HVNL incurs a cost which is about 4,000 times higher than that of Algorithm HHNL when the memory buffer is small (B = 10,000); but in Simulation 13, the cost incurred by Algorithm HHNL is more than 5 times higher than that by Algorithm HVNL. As a result, it is important to choose an appropriate algorithm for a given situation.

2. If the number of documents in one of the two document collections, say M, is originally very small or becomes very small after a selection, then algorithm 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 (it is 70 for simulation 15).

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 algorithm 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 algorithm VVM, these costs have no impact in ranking these algorithms.

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

6.2 An Integrated Algorithm

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 ($B, P, \alpha$) and query parameters ($\lambda$, selectivities of predicates on non-textual attributes). This integrated algorithm can be sketched as follows:

```
If none of the two collections has inverted file /* in this case, only HHNL can be used */
    {compute hhs using formula (HHS1);
     compute bhhs; /* the counterpart of hhs when the backward order is used (formula not shown) */
     If hhs ≤ bhhs, use the forward order of HHNL;
     Else use the backward order of HHNL;
    }
If only one collection has inverted file /* only HHNL and HVNL can be used in this case */
    {If there is no selection
        {compute hhs using formula (HHS1);
         compute bhhs;
         compute hvs using formula (HVS1);
         compute bhvs; /* the counterpart of hvs when the backward order is used (formula not shown) */
        }
    Else
        {estimate the number of documents that can participate in the join using the selectivities;
         compute hhs using formula (HHS2);
         compute bhhs;
         compute hvs using formula (HVS2);
         compute bhvs;
        }
     use the algorithm with the lowest estimated cost;
    }
If both collections have inverted file
    {If there is no selection
        {compute hhs using formula (HHS1);
```

compute *bhhs*;

compute *hvs* using formula (HVS1);

compute *bhvs*; /* the counterpart of *hvs* when the backward order is used (formula not shown) */

compute *vvs* using formula (VVS);

}

Else

{estimate the number of documents that can participate in the join using the selectivities;

compute *hhs* using formula (HHS2);

compute *bhhs*;

compute *hvs* using formula (HVS2);

compute *bhvs*;

compute *vvs* using formula (VVS);

}

use the algorithm with the lowest estimated cost;

}

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 algorithm HVNL can be very competitive only when the number of documents in one of the two document collections is/becomes very small, and algorithm 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, algorithm 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) conduct more detailed simulation and experiment.

Acknowledgments

We would like to thank the anonymous reviewers for their valuable suggestions to improve the paper. This research 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.

References

[1] D. Beech, P. Chellone, and C. Ellis, "An ADT Approach to Full Text". ISO/IEC JTC1/SC21/WG3 DBL CBR-57, 1992.

[2] P. Bernstein, E. Wong, C. Reeve, and J. Rothnie, "Query Processing in a System for Distributed Databases (SDD-1)". ACM Trans. on Database Systems, vol. 6, no. 4, pp. 602-625, December 1981.

[3] C. Buckley, G. Salton, and J. Allan, "Automatic Retrieval with Locality Information Using Smart". Proc. first Text REtrieval Conference, Gaithersburg, pp. 59-72, March 1993.

[4] U. Dayal, and H-Y. Hwang, "View Definition and Generalization for Database Integration in a Multidatabase system". IEEE Trans. on Soft. Engi., vol. 10, no. 6, pp. 628-644, Nov. 1984.

[5] W. Du, R. Krishnamurthy, and M. Shan, "Query Optimization in Heterogeneous Databases". Proc. VLDB Conference, Vancouver, pp. 277-291, August 1992.

[6] S. Dumais, and J. Nielson, "Automating the Assignment of Submitted Manuscripts to Reviewers". Proc. ACM SIGIR Conference, Copenhagen, June 1992.

[7] S. Ghose, "File Organization: The Consecutive Retrieval Property". Comm. of the ACM, vol. 15, no. 9, pp. 802-808, September 1972.

[8] D. Harman, "Overview of the First Text Retrieval Conference" edited by D. Harman, Computer Systems Technology, U.S. Department of Commerce, NIST, 1993.

[9] W. Litwin, L. Mark, and N. Roussopoulis, "Interoperability of Multiple Autonomous Databases". ACM Computing Surveys, vol. 22, no. 3, pp. 267-293, Sept. 1990.

[10] C. Liu, "Introduction to Combinatorial Mathematics". McGraw-Hill, 1968.

[11] L. Lilian, and B. Bhargava, "A Scheme for Batch Verification of Integrity Assertions in a Database System". IEEE Trans. on Soft. Engi., vol. 10, no. 6, pp. 664-680, Nov. 1984.

[12] W. Meng, and C. Yu, "Query Processing in Multidatabase Systems". In Modern Database Systems: The Object Model, Interoperability, and Beyond, edited by W. Kim, Addison-Wesley/ACM Press, 1995, Chapter 27, pp. 551-572.

[13] W. Meng, C. Yu, and W. Kim, "A Theory of Translation from Relational Queries to Hierarchical Queries". IEEE Trans. on Knowl. and Data Engi., vol. 7, no. 2, pp. 228-245, April 1995.

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

[15] L. Saxton and V. Raghavan, "Design of an Integrated Information Retrieval/Database Management System". IEEE Trans. on Knowl. and Data Engi., vol. 2, no. 2, pp. 210-219, June 1990.

[16] A. Sheth, and J. Larson, "Federated Database Systems for Managing Distributed, Heterogeneous, and Autonomous Databases". ACM Computing Surveys, vol. 22, no. 3, pp. 183-236, Sept. 1990.

[17] R. Swaminathan, and D. Wagner, "On the Consecutive-Retrieval Problem". SIAM J. Comput., vol. 23, no. 2, pp. 398-414, April 1994.

[18] A. Tomasic, H. Garcia-Molina, and K. Shoens, "Incremental Updates of Inverted Lists for Text Document Retrieval". Proc. ACM SIGMOD Conference, pp. 289-300, Minneapolis, May 1994.

[19] C. Yu, Y. Zhang, W. Meng, W. Kim, G. Wang, T. Pham, and S. Dao, "Translation of Object-Oriented Queries to Relational Queries". Proc. IEEE Conference on Data Engineering, pp. 90-97, Taipei, Taiwan, March 1995.