Fast Evaluation of Structured Queries for Information Retrieval*

Eric W. Brown

Computer Science Department, University of Massachusetts
Amherst, MA 01003-4610, USA
brown@cs.umass.edu

Abstract

Information retrieval systems are being challenged to manage larger and larger document collections. In an effort to provide better retrieval performance on large collections, more sophisticated retrieval techniques have been developed that support rich, structured queries. Structured queries are not amenable to previously proposed optimization techniques. Optimizing execution, however, is even more important in the context of large document collections. We present a new structured query optimization technique which we have implemented in an inference network-based information retrieval system. Experimental results show that query evaluation time can be reduced by more than half with little impact on retrieval effectiveness.

1 Introduction

Speed has always been an important factor in the success and acceptance of information retrieval systems. If an information retrieval system is too slow it will be intolerable to use, regardless of its ability to identify relevant documents. Recent trends in the volume and availability of information suggest that system speed will only become more important. Commercial document collections already contain tens of gigabytes of data, and projects involving digital libraries forecast document collections containing hundreds of gigabytes of data. As document collections become larger, document retrieval inevitably becomes more expensive. Moreover, more sophisticated retrieval techniques are necessary to identify relevant documents. Unfortunately, more sophisticated retrieval typically implies more expensive retrieval, compounding the problem of providing answers quickly and efficiently.

The goal of our work here is to reduce the cost of evaluating queries in sophisticated retrieval systems. In particular, we are interested in statistical models that support structured queries. An example of such a model is the inference network-based information retrieval model [19], as implemented in the INQUERY full-text information retrieval system [4]. INQUERY has established itself as a solid performer in terms of retrieval effectiveness [6], or the ability to satisfy a user's information need by identifying the documents that contain the desired information. INQUERY's retrieval effectiveness is due to the power of the inference network model, which treats document retrieval as an evidential reasoning process. Evidence from a variety of sources may be combined using structured queries to produce a final probabilistic belief in the relevance of a given document. While the power of this model yields strong retrieval effectiveness, the structured queries supported by the model present a challenge when considering optimization techniques.

We have addressed this challenge by developing an optimization technique applicable to systems that support structured queries. We have implemented and measured our technique in INQUERY and found that query evaluation time is reduced by over 50%. Furthermore, this is accomplished with no noticeable impact on retrieval effectiveness. In the next section, we describe query evaluation in INQUERY. In Section 3, we describe our new optimization technique. Implementation details are considered in Section 4. Section 5 contains a discussion of our experiments. In Section 6 we discuss related work, and in Section 7 we offer concluding remarks. Our contributions include a new optimization technique applicable to information retrieval systems that support structured queries, and an evaluation of the technique using an actual implementation on realistic document collections.

2 Structured Query Evaluation

In INQUERY, a user's information need is satisfied by expressing that need as a query and evaluating the query against a collection of documents. Evaluating the query for a given document produces an estimate of the probability of that document satisfying the information need, expressed as a *final belief score*. After all of the documents in the collection have been evaluated, they are ranked based on their final belief scores. A ranked document list is then returned to the user.

A query consists of *indexed concepts*, *belief operators*, and *proximity operators*. These elements are combined in a tree structure with indexed concepts at the leaves and operators at the internal nodes. An example query is shown in Figure 1, where operators are prefixed with a hash mark (#). An indexed concept is a term or other special object identified at indexing time. A proximity operator produces *constructed concepts* by combining indexed concepts and other constructed concepts at query processing time.[1] Concepts contribute *belief values* for every document in which they appear. Belief operators describe how to combine these belief values to produce the final belief score.

*This work is supported by the National Science Foundation Center for Intelligent Information Retrieval at the University of Massachusetts.

SIGIR'95 Seattle WA USA© 1995 ACM 0-89791-714-6/95/07.$3.50

[1] This definition of concept is a slight departure from the formal definition in the inference network [19]. The distinction between indexed and constructed concepts is introduced here to facilitate discussion from an implementation perspective.

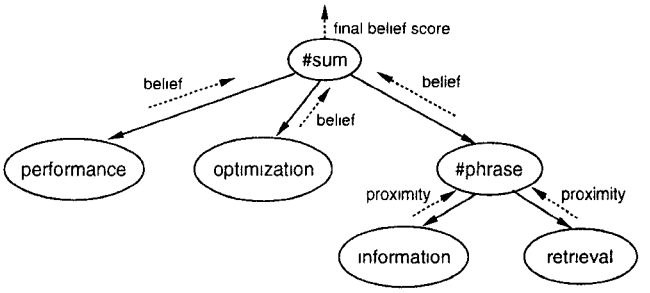

Figure 1: Example query in internal tree form.

Belief operators operate on belief values and return belief values. The belief operators include **and**, **or**, **not**, **sum**, **weighted sum**, and **maximum**. The first three are probabilistic implementations of the traditional boolean operators. The next two return the average and weighted average, respectively, of their children's belief values. The last operator returns the maximum of the belief values from its children.

Proximity operators operate on *proximity lists* and return either a new proximity list or a belief value. A proximity list contains the locations where its associated concept occurs in a given document. For example, in Figure 1 the proximity list for the term "information" in document j would contain the locations of each occurrence of "information" in document j. When the #phrase operator combines that proximity list with the proximity list for the term "retrieval" in document j, a new proximity list for the phrase "information retrieval" is constructed that contains the locations where "information retrieval" appears in document j. This may be returned to a parent proximity operator, or a belief value may be computed from the proximity list and returned to a parent belief operator.

The proximity operators include **phrase**, **ordered distance n**, **unordered window n**, **synonym**, and **passage sum**. The **ordered distance n** operator identifies documents that contain all of the operator's child concepts $\{c_1 \ldots c_k\}$ with the constraint that the concepts must appear in order and be spaced such that the distance between c_i and c_{i+1} is less than or equal to n. The **unordered window n** operator is similar except that all of the child concepts must appear within a window of size n and they may appear in any order. The **phrase** operator is initially evaluated as an **ordered distance n** with $n = 3$. However, depending on the quality of the resultant phrase, the operator may ultimately be evaluated as an **ordered distance n** with $n = 3$, a **sum**, or a **maximum** of these two.

The **synonym** function combines two or more proximity lists into a single proximity list by taking the union of the locations for each document in the lists. The new proximity list represents a constructed concept that occurs anywhere any of the child concepts occur.

The last function, **passage sum**, calculates a belief for a document as follows. First, the document is divided into fixed size overlapping passages, where the last half of each passage overlaps the first half of the subsequent passage. Next, a belief score for each passage is calculated based on the number of occurrences of each of the child concepts within the passage and any weights associated with the child concepts. Finally, the maximum passage belief is returned as the belief for the document. Proximity lists are required from the children to determine concept occurrences within each passage, and a belief list is returned from the passage operator itself.

The belief value contributed by a concept for a given document is calculated using a probabilistic version of the $tf \cdot idf$ score. The *tf* weight is directly proportional to the within document frequency of the concept, such that the more times the concept appears in the document, the greater the belief value. The *idf* weight is inversely proportional to the concept's document count (the number documents in which the concept appears), such that the greater the document count, the smaller the belief value. The details of belief value calculation can be found in the Appendix.

The document counts, within document frequencies, and proximity lists for indexed concepts are extracted and stored in an inverted file [5, 8] when the document collection is indexed. An inverted file consists of a record, or inverted list, for every indexed concept that appears in the document collection. A concept's inverted list contains its document count and an entry for every document in which that concept appears, identifying the document and giving the within document frequency and proximity list of the concept within the document.

To facilitate locating information about a particular document in an inverted list, the document entries are stored in document id order. This naturally leads to the following query processing strategy. First, each node in the query tree is initialized with the next document id (NID) to be processed at that node. For indexed concept (leaf) nodes, this is simply the id of the first document that appears in the inverted list for that concept. Operator (internal) nodes are classified as either union or intersection style operators. Union style operators calculate a result for the current document if *at least one* of its children contributes a result for that document (e.g., **weighted sum**). Intersection style operators calculate a result for the current document only if *all* of its children contribute a result for that document (e.g., **ordered distance n**) A union style operator is initialized with the *minimum* of its children's NIDs, while an intersection style operator is initialized with the *maximum* of its children's NIDs

Processing is performed document-at-a-time with the current document to process determined by the NID at the query tree root The query tree is evaluated in a depth-first fashion for the current document. When a node representing a concept is encountered, a belief value for the current document is computed. The belief values flow from the leaves to the root, being combined according to the belief operators along the way. In addition, as each node is evaluated the node's NID is updated appropriately from its children. When the root node returns the final belief score for the current document, it is saved in a list for later ranking. This process repeats until the NID at the root node indicates that all documents have been processed. The list of final belief scores can then be sorted and the ranked listing returned. Note that the only documents evaluated are those that appear in the inverted lists for the indexed concepts in the query. All other documents receive a default final belief score.

It turns out that an extra query processing step is required. In order to calculate a belief value for a constructed concept (e g , a phrase), we need the concept's *idf* weight. The *idf* weight depends on the number of documents in which the concept occurs. This is unknown until the constructed concept has been evaluated for all of the documents. Therefore, a preprocessing step is needed to fully evaluate the constructed concepts and determine their *idf* weights. The results of this preprocessing step are saved in temporary inverted lists, allowing proximity lists and belief values to be immediately obtained from constructed concepts during the final query evaluation phase.

3 Optimization of Structured Queries

There are two factors that determine the cost of query evaluation. First, there is the complexity of the query. The discussion in Section 2 suggests that queries may be quite complex. The more complex the query, the more processing required for each document in order to evaluate the document's final belief score. The second

factor is the size of the set of documents that must be evaluated, or the *candidate document set*. This set may be quite large. Moffat and Zobel [12] found that for queries containing around 40 terms, using the terms' inverted lists to populate the candidate document set caused nearly 75% of the documents in the collection to be placed in the candidate document set. This is consistent with our results reported below, where our unoptimized candidate document set typically contained over half of the documents in the collection.

Given the relatively small number of top documents a user might actually review in an interactive system, such a large candidate document set seems exorbitant. If our document collection contains one million documents, the system may have to evaluate over five hundred thousand documents, while the user probably won't consider more than the top one thousand documents. Therefore, the goal of our optimization technique is to constrain the set of candidate documents. If we can reduce the size of the candidate document set, we will reduce the number of per document evaluations of the query tree, reducing overall query processing time. Moreover, if we are no longer processing every document that appears in the inverted lists, we may be able to skip portions of inverted lists [13]. If the skipped portions are large enough and our inverted list implementation provides the necessary functionality, the overall number of disk I/Os might be reduced.

To constrain the set of candidate documents, we want to add just those documents that have a strong chance of satisfying the user's information need. Without actually evaluating the query, the best we can do to estimate this chance for a given document is to consider the belief contributions from the indexed concepts in the query. Recall that the belief value for concept i in document j is a product of the *idf* weight for concept i and the *tf* weight for concept i in document j. This leads to the following two observations and corresponding rules:

1. Due to their large *idf* weights, rarely occurring concepts are likely to make large contributions to a document's final belief score. Therefore, they will identify good candidate documents. For a concept whose *idf* weight exceeds some threshold, add to the candidate document set all documents that contain the concept (i.e., all documents that appear in the concept's inverted list).

2. More frequently occurring concepts may still contribute significant belief values for the documents in which they appear frequently (i.e., have a large *tf* weight). For a concept that does not exceed the *idf* weight threshold, add to the candidate document set the documents associated with the concept's top n *tf* weights.

An indexed concept's *idf* weight is inversely proportional to the length of its inverted list. Rather than establish an *idf* weight threshold for candidate set population, we use an inverted list length threshold. An inverted list is *short* if it can be obtained in a single disk read, otherwise it is *long*. From our first rule, all of the documents that appear in a short list will be used to populate the candidate document set. The cost associated with this activity is a single disk read per short inverted list. Since one disk read is required anyway to access an inverted list for later processing, populating the candidate document set with a short list will incur no extra I/O costs.

From our second rule, we need to obtain the documents associated with the top n *tf* weights in the long inverted lists. This suggests that the inverted lists should be sorted by *tf* weight. However, query evaluation is document driven and requires that the inverted lists be sorted by document identifier. Instead, if n is defined to be relatively small, we can maintain a separate list of the documents associated with the top n *tf* weights for each long inverted list. Zipf's Law [21] suggests that there will be relatively few long inverted lists, but they will consume the majority of the space in the inverted file. If each top document list is constrained to be smaller than a disk page, then the overhead associated with the top document lists will be a small percentage of the total space occupied by the long inverted lists. Furthermore, obtaining the top document list for a long inverted list will require a single disk read.

Using our two rules, the candidate document set is created in a final preprocessing pass over the query tree, after the constructed concepts have been built. When an indexed concept with a short list is encountered, all of the documents in that list are added to the candidate set. When an indexed concept with a long list is encountered, the documents with the top n *tf* weights from that list are added to the candidate set. When a constructed concept built by a proximity operator is encountered (e.g., a phrase), it could be handled in the same way as an indexed concept. However, for simplicity in the current implementation, constructed concepts are treated like short lists and all of the documents in a constructed concept's inverted list are added to the candidate set.

One special case is the **not** operator. In this case, we ignore the subtree below the **not** altogether. Theoretically, the **not** would add every document that does not contain the concept represented by its child node. In the probabilistic implementation, the **not** does not increase the belief in documents that do not contain the negated concept, but merely reduces the belief in documents that do contain the negated concept. Therefore, it is sufficient to ignore the **not** when establishing the candidate set and simply evaluate the **not** on the candidate set established from the rest of the query tree.

The final candidate document set is used to drive the document evaluation process. Rather than choose the current document to evaluate based on the NID at the root of the query tree, we simply evaluate each of the documents in the candidate set. Otherwise, query evaluation proceeds as described in Section 2. Each document in the candidate set is fully evaluated and receives an accurate final belief score. The final relative ranking of the documents in the candidate set will be the same as if no optimization had been used. The only difference will be that documents that were not added to the candidate set will receive the default document score and may appear lower in the final ranking than they would have had they been evaluated.

4 Implementation

Our optimization technique places certain functionality requirements on the inverted file implementation. First, we must be able to store the top document lists for the long inverted lists. Second, we must be able to skip through long lists if we are to realize any savings in I/O. Finally, we must be able to distinguish between the different types of lists and handle them accordingly at indexing time, query processing time, and collection modification time.

Fortunately, these functionality requirements are easily met in INQUERY. INQUERY's inverted file is managed by the Mneme persistent object store [14, 2]. Mneme provides storage and retrieval of objects, where an object is a chunk of contiguous bytes that has been assigned a unique identifier. Mneme does not interpret the contents of objects, but does support inter-object references, allowing the fabrication of complex data structures. Mneme is geared towards performance and extensibility. Policies controlling activities such as buffer management, file organization, clustering, and object creation (among others) may be customized for the particular client application. This is particularly important in an inverted file environment where objects will come in a variety of sizes and exhibit unusual access patterns. Customizing the management of these objects will lead to improved overall performance.

Using Mneme our inverted lists are managed as follows. Short lists are defined to be 8 Kbytes or less. This is the size of a disk read in a typical Unix system. Short lists are stored in fixed length

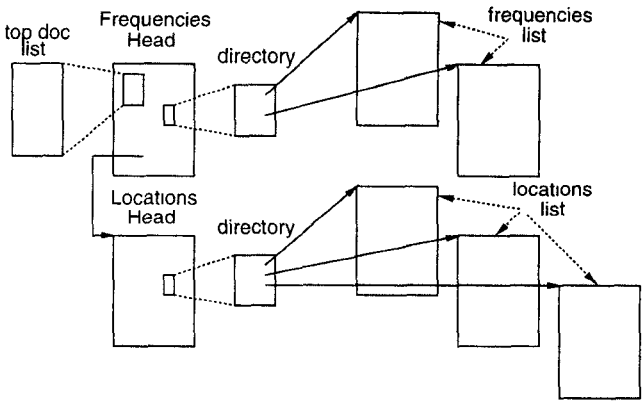

Figure 2: Long inverted list structure.

objects, ranging in size from 16 bytes to 8 Kbytes by powers of 2 (i e , 16, 32, 64, ..., 8 K). A document entry in the list for term t consists of a document identifier j, the frequency of the most frequent term in the document, the within document frequency of the term in the document, and the location of each occurrence of the term within the document.

A short list is compressed in two steps. First, the proximity list associated with each document entry is delta encoded, where the first location is stored as an absolute value and all subsequent locations are stored as deltas from the previous location. This yields numbers of significantly smaller magnitude. Then, all numbers in the list are represented in base 2 using the *minimum* number of bytes (up to four), with a continuation bit reserved in each byte. This results in variable length numbers where the largest representable number is 2^{28}.

Long inverted lists (larger than 8 Kbytes) are stored as shown in Figure 2. A long inverted list is split into two distinct lists: a frequencies list and a locations list. The frequencies list contains the document id and frequency statistics from each of the document records in the original inverted list. The locations list contains the locations (proximity lists) from the document entries. Each of these new lists is stored in 8 Kbyte objects accessed through a directory. A directory entry contains a pointer to an object, along with the document id for the first list entry in the object. To obtain the information for a specific document, the directory is used to identify and directly access the objects that contain the desired information.

The directory for the frequencies list is compressed and stored in a special 8 Kbyte object called the Frequency Head. When the inverted list is first accessed, the Frequency Head is obtained and the directory is decompressed. This is all that is needed to access the frequencies list and satisfy requests for belief values from parent belief operators. If a proximity list is required, the Locations Head must be obtained. The Locations Head is another special 8 Kbyte object that contains the compressed directory for the locations list. Both the frequencies list and the locations list are accessed simultaneously to return the desired proximity list.

The per document proximity lists in a locations list are delta encoded and compressed just as they are in the short inverted lists. In the frequencies list, the document ids are also delta encoded before all of the numbers in the list are compressed using the same compression scheme as for the short lists. Note that the document ids in short inverted lists could be delta encoded as well, although the additional space savings would be small.

The Head objects will store the tails of their respective lists if there is enough room. In addition, the Frequencies Head contains the top document list stored in a compressed format. For our initial implementation, we set the number of top documents n to 1000. Within inverted list i, documents are ranked based on their tf weights, calculated as the normalized term frequency ntf_{ij} (see Appendix). This produces a floating point number between 0.0 and 1.0. To increase the amount of compression possible on the top document list, each document's normalized term frequency was multiplied by 16383 (i.e., $2^{14} - 1$) to produce an integer guaranteed to fit in two bytes or less using our variable length compression technique. This reduces the precision of our within list ranking function, but yields a significant space savings. The lost precision is seen only at the boundary score for the worst document in the top document list, where we may not be sure that we have the best document mapped to that integer. All documents with larger integer scores are guaranteed to have a larger ntf_{ij}.

This implementation provides the functionality necessary to support our optimization technique, including storage of the top document lists and the distinction between short and long inverted lists. Moreover, the directory based access into the large inverted lists supports skipping through the lists and the potential for disk I/O savings.

5 Performance Evaluation

Optimization techniques for ranking information retrieval systems may be classified as either *safe* or *unsafe*. Safe techniques have no impact on retrieval effectiveness, while unsafe techniques may trade retrieval effectiveness for execution speed. The optimization we have proposed is unsafe. Any evaluation of an unsafe optimization technique requires measuring the execution speeds of the base and optimized systems, as well as assessing the impact of the optimization technique on the system's retrieval effectiveness. We describe our evaluation below, including the platform on which we ran our experiments, the test collections and query sets used, the performance measured, and the levels of retrieval effectiveness observed.

5.1 Platform

All of our experiments were run as superuser with logins disabled on an idle DECSystem 3000/600 (Alpha AXP CPU clocked at 175 MHz) running OSF/1 V3.0. The system was configured with 64 Mbytes of main memory, one DEC 1.0 Gbyte RZ26l Winchester SCSI disk, and one DEC 2.0 Gbyte RZ28b Winchester SCSI disk. The executables were compiled with the DEC C compiler driver 3.11 using optimization level 2. All of the data files and executables were stored on the larger local disk, and a 64 Mbyte "chill file" was read before each query processing run to purge the operating system file buffers and guarantee that no inverted file data was cached by the file system across runs. In all cases we allocated 15 Mbytes of Mneme buffer space to cache memory resident inverted list objects.

5.2 Test Collections

For our experiments we used three test collections drawn from the three volume *TIPSTER* document collection used in the *TREC* [6] evaluations. The *TIPSTER* document collection consists of articles and abstracts from various periodicals, Department of Energy abstracts, Associated Press articles, Federal Register articles, and U.S. Patent Office reports. Statistics for the test collections can be found in Table 1, where *Terms* is the number of unique indexed concepts and *Postings* is the total number of occurrences of the indexed concepts. **Tip1** is volume 1, **Tip12** is volumes 1 and 2, and **Tip123** is all three volumes.

The test collections were indexed automatically, using stemming to reduce words to common roots and a stop words list to

Table 1: Test collection statistics.

Collection	Size (MB)	Docs	Terms	Postings
Tip1	1206	510343	639914	112812693
Tip12	2069	741562	859121	191742705
Tip123	3181	1077872	1090896	281417622

Table 2: Inverted file space requirements (Mbytes).

Collection	IL Data	Overheads (% of IL data)			Total
		Top Docs	Free Space	Other	
Tip1	338	22 (6.5)	89 (26 4)	9 (2.7)	458
Tip12	574	30 (5 3)	122 (21 2)	11 (1 9)	737
Tip123	836	39 (4 6)	154 (18.4)	14 (1.7)	1043

eliminate words too frequent to be worth indexing. Feature recognizers were also used to identify city names, company names, foreign country names (i.e., not the United States), and references to the United States. Statistics for the inverted files generated during the indexing process can be found in Table 2. For each file the table gives the size of the inverted list data after compression, the overheads in the file, and the total file size. *Top Docs* is the space required for the top document tables, *Free Space* is unused space at the end of an object that could be allocated in the future, and *Other* is data structure and Mneme overhead. Most of the free space appears in the Head objects of long inverted lists, indicating that a better implementation could be more space efficient. Regardless, the overall inverted files are still only 33%–38% of the size of their respective document collections.

5.3 Query Sets

The query sets used in our experiments were generated locally from topics provided for the *TREC* evaluations. The first query set, **Query Set 1**, was generated from *TIPSTER topics 51–100* using automatic and semi-automatic methods. The resultant fifty queries consisted primarily of weighted sums of terms, phrases, and ordered proximities, with an average of 39 terms per query.

The second query set, **Query Set 2**, was generated from *TIPSTER topics 151–200* in a series of steps. First, a base query set was created using automatic methods. Next, each base query was run against a *PhraseFinder* [9] database built from *TIPSTER* volumes 1 and 2. *PhraseFinder* returns a set of phrases extracted from the supporting database based on the given query. Thirty new phrases were automatically added to each query, forming an augmented query. The augmented queries were then interactively modified to simulate changes an end user might make to automatically generated queries. The changes were limited to the deletion of words judged spurious by the user, changes in weighting based on perceived relative importance, and the addition of proximity constraints. Approximately five minutes were spent on each query. Finally, each modified query was duplicated and one copy was placed inside a passage sum operator with a passage size of 200, which in turn was added to the other copy in a weighted sum. The final set of fifty queries contained an average of 105 terms per query.

5.4 Performance Results

Each experimental configuration involved three variables: query set, document collection, and level of optimization. **Query Set 1** was run against all three document collections, while **Query Set 2** was run against just the first two document collections (relevance judgements were not available for topics 151–200 on volume 3). For a given query set and document collection, performance was measured at three levels of optimization: **all**, **1000**, and **100**. **all** is the unoptimized baseline, where the candidate document set is defined by the original query processing strategy described in Section 2. **1000** is the most conservative level of optimization we considered, where the candidate document set is populated from constructed concepts, short inverted lists, and the top 1000 documents from long inverted lists. **100** is a more aggressive level of optimization, where the candidate document set is populated from constructed concepts, short inverted lists, and the top 100 documents from long inverted lists. The level of optimization is controllable with a run-time switch allowing the same inverted file to be used for all optimization levels within a given configuration

Table 3: Number of documents evaluated.

Collection	Qry Set	Documents (% change)		
		All	**1000**	**100**
Tip1	**1**	13436637	1057900 (−92)	382841 (−97)
	2	11131087	977694 (−91)	419740 (−96)
Tip12	**1**	21207958	1263141 (−94)	559012 (−97)
	2	17384562	1181650 (−93)	611787 (−96)
Tip123	**1**	29763641	1439024 (−95)	710976 (−98)

Table 4. Wall clock times.

Collection	Qry Set	Seconds (% change)		
		All	**1000**	**100**
Tip1	**1**	1364	632 (−54)	569 (−58)
	2	2530	938 (−63)	806 (−68)
Tip12	**1**	2258	1054 (−53)	980 (−57)
	2	4195	1535 (−63)	1394 (−67)
Tip123	**1**	3300	1518 (−54)	1445 (−56)

Our first metric of interest is the size of the candidate document set. Table 3 gives the total number of documents evaluated in each query set configuration. For example, when **Query Set 1** was run against **Tip1** with no optimization, scores were calculated for a total of 13,436,637 documents, or an average of 268,733 documents per query. This is over half of the documents in the entire collection. However, when only the top 1000 documents from long inverted lists are used to populate the candidate document set, scores were calculated for a total of 1,057,900 documents, or an average of 21,158 documents per query. We have reduced the number of documents being evaluated by over 90%. The more aggressive level of optimization reduces the number of documents being evaluated even further. From this table it is clear that we have met our first goal of reducing the size of the candidate document set

The more important question is how this translates into a reduction in query processing time. To answer this question, we measured the real (wall-clock) time required to run each query set configuration. Real time was measured using the GNU time command and includes all time from start to finish of the query set batch run, including the processing of relevance judgements. In Table 4, we report the average real time of ten separate runs for each configuration. In all cases the range between the best and worst times recorded for a given configuration was less than 3.3% of the average for the configuration.

The query processing speedup realized even with our most conservative level of optimization is quite dramatic. In all cases, query processing time is cut at least in half. Moreover, most of the improvement is realized in the more conservative **1000** configuration. Optimizing more aggressively in the **100** configuration yields just

an additional 2%–5% improvement over the baseline. Clearly we have achieved our ultimate goal of reducing query processing time

With the candidate document set considerably reduced, we would expect to be able to skip significant portions of the long inverted lists during query evaluation. To measure this, we counted the number of whole objects skipped during long inverted list processing Perhaps surprisingly, in all **1000** configurations there was no increase in the number of long list objects skipped. In fact, even at more aggressive optimization levels, the number of additional objects skipped was minimal. Moreover, the real impact of any skipping was measured in terms of a reduction in the number of object faults, where an object fault occurs when a non-memory resident object is accessed and must be read from disk. Even when there was an increase in skipping, the reduction in object faults was insignificant, indicating that we were skipping memory resident objects which wouldn't have required a disk read anyway.

The reason for this effect is twofold. First, the information in the long inverted lists is very densely packed in order of document id. Second, the membership of the candidate set is independent of document id, meaning the entries in a long inverted list that must be accessed during query processing should be arbitrarily distributed over the entire list. Therefore, even though we are in fact skipping large portions of the long lists, we still end up accessing at least one document entry in nearly every object in the lists.

To investigate this effect further, we built our inverted files using 2 Kbyte objects in the frequencies and locations lists. In this version skipping was more noticeable (especially at more aggressive optimization levels), but again the number of object faults was reduced by less than 2%. Moreover, since disk reads are 8 Kbytes, we wouldn't expect to see any reduction in the number of raw disk I/Os when compared with the version that used 8 Kbyte objects.

5.5 Retrieval Effectiveness

Along with query processing speed, we must also look at the impact on retrieval effectiveness in order to fully evaluate our unsafe optimization technique. Precision at standard recall points obtained with different levels of optimization for each of our five query set/document collection combinations is reported in Tables 5– 9. The relevance judgements used to generate these tables came from the *TREC* evaluations. We show precision based on full rankings at the standard 11 recall points and the 11pt average, where precision at 0% recall is interpolated. As before, **all** is the unoptimized baseline version, while **1000** through **50** are optimized versions where the label indicates the number of top documents taken from long inverted lists to populate the candidate document set. We show a broader range of optimization levels here than in our timing test to give a better feel for the impact on retrieval effectiveness as the optimization becomes more aggressive. In each of the tables, percent change is from the baseline version.

Consider the results for the **1000** configuration in Tables 5–9. For all query set/document collection combinations, retrieval effectiveness is remarkably good. At recall levels up to 70%, there is no noticeable degradation in precision The implication here is that the high end of a document ranking returned by the optimized system, or the documents most likely to be considered by a user in an interactive system, will be just as rich in relevant documents as in the unoptimized version. Furthermore, the 11pt averages are not significantly different from those for the unoptimized version.

Now consider the results in Table 5. As the optimization becomes more aggressive (from **1000** to **50**), we see two trends. First, at low recall, precision actually improves somewhat and then falls off. This indicates that the technique is doing a good job of identifying the very best candidate documents, and is consistent with other results using similar techniques [16, 12]. Second, at high recall, precision becomes significantly worse as the optimization becomes more aggressive. This is because we are not considering documents which have a strong combined belief from all of the query terms, but lack a single query term belief strong enough to place the document in the candidate set.

In Tables 8 and 9 we do not see any improvement in precision at low recall as the optimization becomes more aggressive. This is due to the use of the **passage sum** operator in **Query Set 2**. The calculation of belief for concept i in document j is slightly modified inside a passage operator since it is based on a passage of the document, rather than the entire document. Thus, our ranking of document j within the inverted list for concept i is slightly inaccurate with respect to the passage operator. This suggests that our retrieval performance could even be improved.

6 Related Work

Some of the earliest optimization work in information retrieval was carried out by Smeaton and van Rijsbergen [18] in the context of a nearest neighbor retrieval model. They describe how an upper bound on the similarity of any unseen document can be calculated based on the unprocessed query terms. If this upper bound is less than the similarity of the current best document, processing may stop. Perry and Willett [15] show how the upper bound technique can be applied to the same model extended to support incremental document score accumulation.

Using a system that returns a full ranking of the document collection, Buckley and Lewit [3], and later Lucarella [11], describe a technique to eliminate processing of entire inverted lists, assuming we are interested in only the top n documents. After processing a given term, the documents can be ranked by their currently accumulated scores, establishing the current set of top n documents. An upper bound on the increase of any document's score can be calculated from the unprocessed terms in the query, assuming the maximum possible $tf \cdot idf$ contribution from each of those terms. If the $n+1^{st}$ document's score plus the upper bound increase is less than the n^{th} document's score, then the set of top n documents has been found. This work and the work on nearest neighbor models forms the basis of our first rule for candidate set population.

There are two variations on the previous scheme, both of which share our goal of constraining the candidate document set. The first variation, proposed by Harman and Candela [7], is called *pruning*. Rather than place a limit on the number of documents returned to the user, they establish an insertion threshold for placing new documents in the candidate set. In order to place a new document in the candidate set, a term's potential score contribution must exceed some threshold Inverted list processing has two distinct phases. First, during a disjunctive phase, documents are added to the candidate set and partial scores are updated. Then, after the insertion threshold is reached, a conjunctive phase occurs where terms are not allowed to add new documents, only update the scores of existing documents.

The second variation was proposed by Moffat and Zobel [12] Rather than use an insertion threshold related to a term's potential score contribution, a hard limit is placed on the size of the candidate document set. The disjunctive phase proceeds until the candidate set is full. Then the conjunctive phase proceeds until all of the query terms have been processed. Both of these schemes populate the candidate document set using a variation of our first rule based on *idf* weight. They do not, however, incorporate a clear version of our second rule based on *tf* weight.

The previous techniques use inverted lists sorted by document id. Other techniques sort inverted lists by *tf* weight. Wong and Lee [20] partition their inverted lists into pages and process the pages in order of upper bound contribution to document score. They describe a number of estimation techniques for determining

Table 5: Precision at standard recall points for **Tip1, Query Set 1.**

	Precision (% change) – 50 queries									
Recall	**all**	**1000**		**500**		**300**		**100**		**50**	
0	83.5	83 7	(+0.2)	83.9	(+0.5)	83.9	(+0.5)	83.9	(+0 5)	83.9	(+0.5)
10	60.3	60.5	(+0 2)	60.9	(+1.0)	60 8	(+0.8)	61 4	(+1.7)	61.6	(+2.1)
20	52 7	53.0	(+0.6)	53.3	(+1.2)	53.4	(+1 3)	53 5	(+1.5)	53 1	(+0.8)
30	46.8	47.1	(+0.6)	47.0	(+0.4)	46.7	(−0.3)	46 1	(−1.7)	44 3	(−5.5)
40	40.6	40.9	(+0.7)	40.9	(+0.8)	41.0	(+1 0)	38.9	(−4.2)	38 4	(−5.5)
50	34.9	35.2	(+1.0)	35.1	(+0.8)	35.1	(+0.7)	33 1	(−5.0)	32 4	(−7 1)
60	30.4	30.6	(+0.6)	30.7	(+1 1)	30 1	(−1.0)	28.1	(−7 6)	27 2	(−10.4)
70	25 3	25.7	(+1.7)	25.6	(+1.3)	24.0	(−5 1)	20 9	(−17 1)	20.4	(−19 4)
80	19 9	19.8	(−0.1)	18 3	(−7 9)	17.7	(−10.9)	15.8	(−20.7)	14.6	(−26 4)
90	12.1	11.6	(−4.6)	11.3	(−6.9)	9.6	(−20.9)	8.6	(−29.4)	7.6	(−37 1)
100	2.4	1.7	(−29 2)	1 5	(−38.7)	1.6	(−36.2)	1 6	(−36.1)	1.6	(−33.3)
average	37.2	37.3	(+0.2)	37 2	(−0 1)	36.7	(−1.2)	35.6	(−4 2)	35.0	(−5.8)

Table 6: Precision at standard recall points for **Tip12, Query Set 1.**

	Precision (% change) – 50 queries									
Recall	**all**	**1000**		**500**		**300**		**100**		**50**	
0	83 6	83.7	(+0.1)	83.5	(−0.1)	83.3	(−0.4)	83 3	(−0.3)	83 6	(+0 0)
10	57.2	57.5	(+0.6)	57.7	(+0.9)	57.7	(+0.9)	56.8	(−0.6)	56 5	(−1.2)
20	49 0	49.5	(+1 0)	49.7	(+1.4)	49.6	(+1 1)	48.7	(−0.7)	48 1	(−1.9)
30	43.1	43.4	(+0.8)	43.5	(+0.9)	43.2	(+0 4)	42.0	(−2.5)	40.3	(−6.4)
40	37.7	38.1	(+1.0)	38 0	(+0.9)	37.3	(−1.0)	34 9	(−7.5)	34 4	(−8 8)
50	32.4	32.9	(+1.5)	32.5	(+0.3)	32.0	(−1.3)	29.2	(−9.8)	28.7	(−11 3)
60	27.7	27 9	(+0 6)	27.2	(−1.8)	26 0	(−6.1)	23.9	(−13.6)	23.2	(−16.5)
70	22.5	22.8	(+1.4)	21.7	(−3.8)	20.2	(−10 4)	17.7	(−21.5)	17.2	(−23.7)
80	17 3	17.0	(−1.6)	15.0	(−13.4)	13 8	(−20.0)	12.2	(−29.3)	12.1	(−29.9)
90	11.2	10.0	(−10.6)	8.5	(−24.2)	7.8	(−30 3)	7.8	(−30.7)	7.8	(−30.1)
100	1.2	0.5	(−59.3)	0.6	(−54 0)	0 6	(−55 4)	0.7	(−47.5)	0.7	(−42.2)
average	34.8	34.9	(+0.1)	34.3	(−1.3)	33.8	(−3.0)	32.5	(−6.7)	32.1	(−7.9)

how many $tf \cdot idf$ weights must be processed to achieve a given level of retrieval effectiveness.

Persin [16] uses thresholds to determine how a $tf \cdot idf$ weight is processed. If the document for the current weight is not in the candidate document set, an insertion threshold is used to determine if the weight justifies adding the document to the candidate set. If the document is already in the candidate set, an addition threshold is used to determine if the weight should modify the document's current score. The addition threshold allows processing of an inverted list to stop as soon as its $tf \cdot idf$ weights fall below the threshold. The insertion threshold ensures that we consider only documents that receive a significant weight contribution from the terms.

Again, the last two schemes attempt to constrain the candidate document set and use some version of our first candidate set population rule. Additionally, they suggest our second rule based on tf weights. Unfortunately, their inverted list organization does not support document based query evaluation, as is necessary for proximity operators. Moreover, document collection updates are very difficult to support when inverted lists are sorted by tf weight.

None of the above ranking system techniques are directly applicable to structured queries. They are all designed for flat queries, and can make assumptions about document score to dynamically constrain the set of candidate documents as query evaluation proceeds. While these techniques might be applicable to certain operators in a structured query tree, it is not clear that applying them in a localized fashion is appropriate. Optimization should be coordinated throughout the query tree. By pre-computing our candidate document set, we ensure that this is the case.

The process of identifying a candidate document set followed by evaluating the query for just those documents is similar in spirit to the two stage query evaluation strategy of the SPIDER information retrieval system [17, 10]. In SPIDER, a signature file is used to identify documents that potentially match the query, and an upper bound is calculated for each document's similarity to the query. Non-inverted document descriptions are then retrieved for these documents in order of best upper bound similarity and used to compute an exact similarity measure. As soon as a document's exact similarity measure exceeds all other documents' upper bound (or exact) similarity measures, this document can be returned as the best matching document. It is possible that a similar signature file scheme could be used to identify our candidate document set, although it would be difficult to calculate reasonable upper bounds for document beliefs from the signature information.

7 Conclusions

We have developed an optimization technique for structured queries that provides a significant execution performance improvement by reducing the number of documents that must be evaluated. Our technique has been implemented in the INQUERY full-text information retrieval system and evaluated using large, realistic document collections. Experimental results show that our optimization can reduce query processing time by over 50% with no noticeable degradation in precision until better than 70% recall. In an interactive system, our optimization is unlikely to impact the user's perception of the effectiveness of the system. However, the reduction in query processing time by more than half is certain to impact the user's perception of the usefulness of the system. Moreover, the level of optimization is tunable at run-time, allowing the user to control the tradeoff between speed and precision.

A key component of our optimization technique is our inverted list implementation, which supports candidate set population activities and provides opportunities for disk I/O savings. While our results indicate that reduced disk I/O is not a significant contributor to the performance improvement realized with our current optimization, there are other safe optimizations that we do not consider here that can take advantage of our inverted list implementation. One example is a boolean style intersection optimization when process-

Table 7: Precision at standard recall points for **Tip123, Query Set 1.**

	Precision (% change) – 50 queries									
Recall	**all**	**1000**		**500**		**300**		**100**		**50**	
0	84 3	84.3	(+0.0)	84.2	(−0.1)	84.1	(−0.2)	84 7	(+0.5)	84.5	(+0.3)
10	54.6	54.8	(+0.4)	55.0	(+0.8)	54.8	(+0.4)	53 6	(−1 8)	53.2	(−2.5)
20	47 1	47 3	(+0.5)	47.3	(+0.4)	47.0	(−0.2)	45.4	(−3 5)	43.9	(−6 6)
30	40.6	40.9	(+0.7)	40.5	(−0.3)	39.7	(−2.3)	37 2	(−8 4)	36 4	(−10 3)
40	35 3	35 5	(+0.5)	34.9	(−1.3)	33.4	(−5.4)	30.7	(−12.9)	30.0	(−14.9)
50	30.3	30.4	(+0.6)	29 5	(−2 6)	27 9	(−7.9)	25 7	(−15 0)	25 7	(−15.2)
60	25 7	25.7	(+0.1)	24 0	(−6.7)	22 4	(−13.0)	21 1	(−18.0)	20 9	(−18.5)
70	20.7	20.0	(−3.6)	18.1	(−12.6)	17.2	(−17 2)	16.7	(−19 2)	16 6	(−19 9)
80	15.5	13.3	(−14.3)	11.8	(−24.1)	11.3	(−27.0)	10.2	(−34 0)	10.2	(−34.2)
90	9.1	7.2	(−20.9)	6.1	(−32 6)	5 8	(−35.9)	6 0	(−34.2)	6.1	(−33 4)
100	0 5	0 2	(−67.2)	0.1	(−73.6)	0.1	(−73.2)	0.1	(−72 7)	0.1	(−72.6)
average	33.1	32.7	(−1.1)	31.9	(−3 4)	31.2	(−5.5)	30 1	(−8.8)	29.8	(−9.9)

Table 8: Precision at standard recall points for **Tip1, Query Set 2.**

	Precision (% change) – 50 queries									
Recall	**all**	**1000**		**500**		**300**		**100**		**50**	
0	91.1	91.1	(+0.0)	91.1	(+0.0)	91.1	(+0.0)	91.1	(+0 0)	91 1	(+0 0)
10	75.9	75.9	(−0.0)	75.8	(−0.1)	75 8	(−0.1)	75 8	(−0 1)	75 8	(−0 1)
20	66.0	66.0	(−0.0)	65.9	(−0 1)	65.8	(−0.3)	65.8	(−0 3)	65 8	(−0 3)
30	55.6	55.6	(+0.1)	55.2	(−0.7)	55 2	(−0.7)	55 0	(−1.1)	54.9	(−1.3)
40	47.4	47.4	(+0.1)	47.0	(−0.8)	47.0	(−0 8)	46.6	(−1 6)	46.4	(−2.1)
50	41.4	41.3	(−0.2)	41 1	(−0.7)	40.8	(−1.3)	40.3	(−2 7)	40 2	(−2 9)
60	35.1	35.0	(−0.1)	34.8	(−0.8)	34.5	(−1.7)	32.8	(−6.4)	32 4	(−7 5)
70	27 4	27.3	(−0 4)	26 6	(−2.9)	26.1	(−5.0)	24 7	(−9 8)	24 6	(−10 2)
80	22 1	21 8	(−1.2)	21.4	(−3.1)	21.4	(−3.1)	19.4	(−12.3)	18.6	(−15 5)
90	15.5	15.3	(−1.5)	14.6	(−6.1)	13.1	(−15.9)	11.0	(−29.4)	10 6	(−31 8)
100	3 7	2.7	(−27.4)	2.3	(−37.0)	1.8	(−51.9)	1.4	(−62.8)	1.3	(−64.6)
average	43.7	43.6	(−0.4)	43.3	(−1.1)	43.0	(−1.8)	42.2	(−3.6)	42.0	(−4 0)

ıng proximity operators, which can provide additional inverted list skipping opportunities. Another example is the specialized access of inverted list information enabled by the separation of frequency statistics from locations. This allows belief operators to avoid the overhead of processing location information in the inverted lists.

Furthermore, our inverted list implementation will easily support updates to the document collection. When new documents are added, we must be able to grow the inverted lists for the indexed concepts that appear in the new documents. This process was described by Brown et al. in [1] using an inverted file structure similar to that described here. The main extensions made here are the separation of frequency and location information in the long inverted lists and the use of directories into the long list objects. The directories also facilitate document deletion, providing direct access to the portion of an inverted list containing the document entry that must be deleted.

Finally, while our implementation and experimental evaluation have been carried out in the context of the inference network model, our technique is generally applicable to any statistical retrieval model that supports structured queries. As these retrieval models are applied to larger and larger document collections, optimization techniques such as ours will become ever more crucial to the success of these systems.

Acknowledgements

Thanks go to Jamie Callan for his helpful comments on early drafts of this paper, Bruce Croft for his guidance throughout this work, and the staff and students of CIIR for their work on the systems used here.

Appendix

The belief value for concept i in document j is calculated with the following formula:

$$belief_{ij} = C + (1 - C)\, ntf_{ij}\, nidf_i \qquad (1)$$

where

$$ntf_{ij} = Ks + (1 - K)\left(\frac{\log(tf_{ij} + 0.5)}{\log(max\_tf_j + 1.0)}\right)$$

$$nidf_i = \frac{\log((N + 0.5)/n_i)}{\log(N + 1.0)}$$

ntf_{ij}	is the normalized within document frequency
$nidf_i$	is the normalized inverse document frequency
tf_{ij}	is the within document frequency
max_tf_j	is the maximum of $\{tf_{1j}, tf_{2j}, \ldots\}$
N	is the # documents in the collection
n_i	is the # documents in which concept i appears

The constants C and K both default to 0.4 in INQUERY, although they may be specified by the user. C is the default belief value returned for documents that do not contain the given concept. K acts to increase the significance of even a single occurrence of a concept in a document. s is used to reduce the influence of document length for long documents. If max_tf_j is greater than 200, then s is set to $200/max\_tf_j$. Otherwise, s is set to 1.0. Additionally, if tf_{ij} is equal to max_tf_j, then ntf_{ij} is set to 1.0.

References

[1] E. W. Brown, J. P. Callan, and W. B. Croft. Fast incremental indexing for full-text information retrieval. In *Proc. of the 20th Inter. Conf. on Very Large Databases (VLDB)*, pages 192–202, Santiago, Sept. 1994.

Table 9: Precision at standard recall points for **Tip12, Query Set 2.**

	Precision (% change) – 50 queries									
Recall	**all**	**1000**		**500**		**300**		**100**		**50**	
0	89.4	89.4	(+0.0)	89 4	(+0.0)	89.4	(+0 0)	89 4	(+0 0)	89.4	(+0.0)
10	73.8	73.8	(−0.0)	73 7	(−0 1)	73.7	(−0 1)	73.7	(−0 1)	73.7	(−0.2)
20	64.3	64.2	(−0 1)	64.1	(−0.3)	64.1	(−0 2)	64.1	(−0.3)	64.1	(−0 3)
30	56.6	56.5	(−0.0)	56.4	(−0.2)	56.2	(−0.5)	56 0	(−1.0)	55 9	(−1.2)
40	49 6	49 6	(−0.0)	49.6	(−0.1)	49.5	(−0.4)	49.0	(−1 2)	48 9	(−1 4)
50	43 6	43.5	(−0.3)	43.2	(−0.9)	43.0	(−1.5)	41 9	(−3 9)	41.8	(−4 2)
60	36 9	36.4	(−1.2)	35.7	(−3.2)	34.7	(−5.8)	33.3	(−9.7)	33 2	(−9 8)
70	30.1	29.5	(−2.1)	28.8	(−4.3)	28.1	(−6 6)	25.8	(−14 5)	25.3	(−16 0)
80	24 7	24.0	(−3.2)	23.4	(−5.2)	21.1	(−14.8)	20.1	(−18.8)	20 0	(−19 4)
90	16.5	15.4	(−6.7)	13.0	(−21.3)	12.7	(−22.9)	12.5	(−24 2)	12 5	(−24 2)
100	2.3	1.4	(−37.9)	1.4	(−39.4)	1.5	(−37.6)	1.0	(−56.4)	1.0	(−55.9)
average	44.4	44.0	(−0 8)	43.5	(−1.8)	43.1	(−2.8)	42 4	(−4 3)	42.3	(−4 5)

[2] E. W. Brown, J. P. Callan, W. B. Croft, and J. E. B. Moss. Supporting full-text information retrieval with a persistent object store. In *Proc. of the 4th Inter. Conf. on Extending Database Technology (EDBT)*, pages 365–378, Cambridge, UK, Mar. 1994

[3] C. Buckley and A. F. Lewit. Optimization of inverted vector searches. In *Proc. of the 8th Inter. ACM SIGIR Conf. on Research and Development in Information Retrieval*, pages 97–110, June 1985.

[4] J. P. Callan, W. B. Croft, and S. M. Harding. The INQUERY retrieval system. In *Proc. of the 3rd Inter. Conf. on Database and Expert Systems Applications*, Sept. 1992.

[5] C. Faloutsos. Access methods for text. *ACM Comput. Surv.*, 17:50–74, 1985.

[6] D. Harman, editor. *The Second Text REtrieval Conference (TREC2)*, Gaithersburg, MD, 1994. National Institute of Standards and Technology Special Publication 500-215.

[7] D. Harman and G. Candela. Retrieving records from a gigabyte of text on a minicomputer using statistical ranking. *J. Amer. Soc. Inf. Sci.*, 41(8):581–589, Dec. 1990.

[8] D. Harman, E. Fox, R. Baeza-Yates, and W. Lee. Inverted files. In W. B. Frakes and R. Baeza-Yates, editors, *Information Retrieval: Data Structures & Algorithms*, chapter 3, pages 28–43. Prentice Hall, Englewood Cliffs, NJ, 1992.

[9] Y. Jing and W. B. Croft. An association thesaurus for information retrieval. In *Proc. of RIAO 94 Conf.*, pages 146–160, New York, Oct. 1994.

[10] D. Knaus and P. Schäuble. Effective and efficient retrieval from large and dynamic document collections. In Harman [6], pages 163–170.

[11] D. Lucarella. A document retrieval system based on nearest neighbour searching. *J. Inf. Sci.*, 14(1):25–33, 1988.

[12] A. Moffat and J. Zobel. Fast ranking in limited space. In *Proc. 10th IEEE Inter. Conf. on Data Engineering*, pages 428–437, Feb. 1994.

[13] A. Moffat and J. Zobel. Self-indexing inverted files for fast text retrieval. Technical Report 94/2, Collaborative Information Technology Research Institute, Department of Computer Science, Royal Melbourne Institute of Technology, Australia, Feb. 1994.

[14] J. E. B. Moss. Design of the Mneme persistent object store. *ACM Trans. Inf. Syst.*, 8(2):103–139, Apr. 1990.

[15] S. A. Perry and P. Willett. A review of the use of inverted files for best match searching in information retrieval systems. *J. Inf. Sci.*, 6(2-3):59–66, 1983.

[16] M. Persin. Document filtering for fast ranking. In *Proc. of the 17th Inter ACM SIGIR Conf. on Research and Development in Information Retrieval*, pages 339–348, Dublin, July 1994.

[17] P. Schäuble. SPIDER: A multiuser information retrieval system for semistructured and dynamic data. In *Proc of the 16th Inter. ACM SIGIR Conf. on Research and Development in Information Retrieval*, pages 318–327, Pittsburgh, June 1993.

[18] A. F. Smeaton and C. J. van Rijsbergen. The nearest neighbour problem in information retrieval. An algorithm using upperbounds. In *Proc. of the 4th Inter. ACM SIGIR Conf. on Research and Development in Information Retrieval*, pages 83–87, Oakland, CA, 1981.

[19] H. R. Turtle and W. B. Croft. Evaluation of an inference network-based retrieval model *ACM Trans. Inf. Syst.*, 9(3):187–222, July 1991.

[20] W. Y. P. Wong and D. L. Lee. Implementations of partial document ranking using inverted files. *Inf. Process. & Mgmnt.*, 29(5):647–669, 1993.

[21] G. K. Zipf. *Human Behavior and the Principle of Least Effort*. Addison-Wesley Press, 1949.