An Approach for Improving Execution Performance in Inference Network Based Information Retrieval*

Eric W. Brown

Technical Report 94–73
September 1994

Department of Computer Science
University of Massachusetts
Amherst, MA 01003
USA

Abstract

The inference network retrieval model provides the ability to combine a variety of retrieval strategies expressed in a rich query language. While this power yields impressive retrieval effectiveness, it also presents barriers to the incorporation of traditional optimization techniques intended to improve the execution efficiency, or speed, of retrieval. The essence of these optimization techniques is the ability to identify the indexing information that will be most useful for evaluating a query, combined with the ability to access from disk just that information. I survey a variety of techniques intended to identify useful indexing information and propose a new strategy for incorporating these techniques into the inference network retrieval model. Additionally, I describe an architecture based on a persistent object store that provides sophisticated management of the indexing data, allowing just the desired indexing data to be accessed from disk during retrieval.

*This work is supported by the National Science Foundation Center for Intelligent Information Retrieval at the University of Massachusetts. Email: brown@cs.umass.edu.

1 Introduction

The human race has a long history of putting things in writing. In our efforts to communicate, record, and document, we inevitably generate some written piece of work that contains history, facts, plans, or information in general. While generating these works, which I shall call documents, is relatively easy, making use of the information contained in those documents can be quite difficult, particularly if there are many documents in existence and we are interested in only a few. It is exactly this functionality, however, that is most critical. The attorney must be able to identify past cases relevant to the current case. The scientist must be able to identify technical papers related to their research. The accountant must be able to locate applicable tax laws. The contractor must be able to find standards and specifications related to the project. Anyone interested in a current event must be able to find relevant news articles. A wealth of documents is useless if we cannot identify the ones that contain the information we need.

One of the first solutions to this problem appeared nearly four thousand years ago when catalogues of documents in libraries were created to aid in keeping track of those documents. More recently, in the 16th century, primitive *indexes* for documents were created. An index is a list of certain keywords or topics. Each entry in the list contains pointers into the documents where descriptions and discussions of the respective keyword or topic may be found. Deciding what keywords and topics should go into an index and which discussions are worthy of an index pointer is a tedious and subjective human task prone to omissions. A shortcoming of an index or catalogue is that an information search must be based on either the keywords actually indexed or the attributes used in the catalogue (e.g., author, title, subject).

To address this limitation, *concordances* were created. A concordance for a collection of documents contains an entry for every term that appears in the collection. A term's entry contains a pointer to every occurrence of that term. Now, an information search was no longer restricted by predetermined keywords or attributes. However, a concordance for a large document such as the Bible might require a good portion of a lifetime to construct by hand, and in one case, the effort involved in such a task is believed to have lead to the insanity of the concordance compiler [58].

With the advent of the computer age in the latter half of the 20th century, concordance construction could be automated, greatly simplifying the task. What used to take years could now be accomplished in minutes. In spite of being relatively complete and simple to construct, a concordance still provided a rather unsophisticated solution to our original problem. Trying to locate information in a large collection of documents using a concordance can be an exercise in frustration, leading to the retrieval of many unrelated documents that just happen to contain terms that we believe are indicative of the information we seek. A more intelligent solution to the problem at hand was still needed.

Over thirty years ago, work towards this intelligent solution began with the birth of *information retrieval* systems. The task of an information retrieval (IR) system is to satisfy a user's information need by identifying the documents in a collection of documents that contain the desired information. This task is accomplished through the use of three basic components: a document representation, a query representation, and a measure of similarity between queries and documents. The document representation provides a formal description of the information contained in the documents, the query representation provides a formal description of the information need, and the similarity measure defines the rules and procedures for matching the information need with the documents that satisfy that need.

Much of the research to date in IR systems has focussed on how to implement these three components for best performance. In the IR community, "performance" is understood to mean retrieval effectiveness, rather than execution efficiency. Retrieval effectiveness refers to an IR system's ability to correctly identify the documents that are relevant to a given query, and is measured in terms of recall and precision. Recall is the percentage of the relevant documents actually identified by the system, and precision is the percentage of the documents identified that are relevant. For example, assume that R documents in the collection are relevant to a query, and D documents are returned by the IR system in response to the query. If r of the D

documents are actually relevant (i.e., from the R relevant documents), then the recall is r/R and the precision is r/D.

One of the more powerful retrieval models to emerge from this research is the inference network retrieval model [56]. This model treats information retrieval as an evidential reasoning process. Using a Baysien inference network, evidence about a document's relevance to a query is combined from different sources to produce a final relevance judgement. While much work has been done to improve the retrieval effectiveness of this model, comparatively little work has been published on improving the execution efficiency, or speed of retrieval, in systems that implement this model. Execution efficiency, however, is of paramount importance. Modern IR systems are challenged with the task of managing document collections containing tens or hundreds of gigabytes of text, and access to these collections must be provided in an interactive environment. If an IR system is to be at all useful in such an environment, execution efficiency issues must be addressed.

Here, then, is the basic question that I propose to answer in this thesis:

How can the execution efficiency of the inference network retrieval model be improved?

My answer to this question is based on the following observation. The execution efficiency of an IR system depends heavily on the document indexing structures used by the system. In the case of the inference network model, as well as many other IR systems, the document indexing structure is an inverted file. An inverted file is typically large, ranging in size from 10% to 40% of the size of the raw document collection. It can easily exceed main memory capacity, such that most of the inverted file must remain on disk while only a small fraction of the file is manipulated in main memory. Much of the cost of performing document retrieval will be in reading the inverted file from disk. If the amount of data that must be read from disk can be reduced, the execution efficiency of the system will be improved.

To guide the effort of reducing disk reads, I introduce the strategy of *maximizing return on I/O*. Return on I/O (ROIO) in an IR system is the amount of progress made towards the task at hand as a result of performing I/O. For example, consider the task of query processing where we wish to identify the top N relevant documents in the collection. Assume we have already performed a certain amount of work and have a set of top N documents ranked in order of estimated relevance. If we then read in 1 Mbyte of indexing structures from disk which neither changes the membership of the set of N documents nor alters their rank, then the ROIO is very low. If instead we read in 8 Kbytes of indexing structures from disk which changes the set of N documents such that recall and precision are improved, then the ROIO is quite high.

This approach is motivated by the skewed distribution of sizes of the data objects being manipulated and the relative inefficiency of I/O operations compared to main memory and CPU operations. The objects in an inverted file, or the inverted lists, have a skewed size distribution due to the Zipfian [62] distribution of term frequencies in a document collection. For a multi-gigabyte document collection, the inverted lists will range in size from a few bytes to multiple megabytes. In practice, the larger inverted lists are accessed quite often during query processing. It is unlikely, however, that all of a large inverted list will contribute significantly to resolving a query. Reading these large inverted lists from disk is quite expensive, representing a significant portion of the total query processing cost. If we can devise techniques that eliminate reading from disk much of a large inverted list, without compromising retrieval effectiveness, then significant savings can be realized.

The two basic requirements of this approach are the ability to identify the indexing data that will provide the most useful information, and the ability to access just that data from the inverted file. My first contribution will be to discover query optimization techniques that allow the most useful indexing data to be identified. Many techniques have been proposed in the IR literature for less sophisticated retrieval

models. These techniques are not directly applicable to the inference network model due to the richness of the query language supported by the model and its ability to combine different retrieval paradigms into a single retrieval framework. I will describe a strategy which allows query optimization techniques to be incorporated into the inference network model.

My second contribution will be to describe an inverted file implementation that provides the ability to access just that data identified as most useful by the query optimization techniques. I claim that the functionality requirements of the inverted file are best met by an advanced data management system, such as a persistent object store. This claim is somewhat counter to the popular belief that the inverted file component of an IR system must be custom built for best performance. I will show that superior performance can be achieved with a persistent object store and that the functionality provided by the persistent object store is necessary to support the optimization techniques I will pursue. I will integrate the INQUERY inference network based document retrieval system [9] with the Mneme persistent object store [40] and demonstrate the execution performance improvements possible with my approach.

A proper investigation of execution efficiency requires representative document collections with which to evaluate suggested optimization techniques. Moreover, for any efficiency claims to be valid, it must be shown that retrieval effectiveness has been maintained. This, however, requires relevance scores for queries and collections, which can only be obtained through subjective, human interaction. Historically, large, realistic collections and queries with proper relevance scores have not been available to researchers. The standard document collections used for IR system evaluation include Cranfield [11], CACM [21], and NPL [52]. All of these collections are less than 10 Mbytes of raw text and are considered tiny compared to current and anticipated IR system requirements.

These small research collections simply do not challenge the physical capacities of today's modern computers. CPUs are getting faster, main memories are becoming larger, disks are becoming larger and faster, and everything is becoming cheaper. Simultaneously, user demands for IR system capacity continue to out pace advances in computer hardware. Thus, it has been difficult to draw conclusions from research into execution efficiency issues using these small collections, and claims of scalability that come out of these results are tenuous at best.

Fortunately, just within the last few years, larger and more realistic collections have become available to IR researchers. For example, the TREC evaluations [24] currently involve 3 Gbytes of raw text and provide queries and relevance scores for retrieval and routing, allowing IR system evaluations and comparisons on relatively large document collections. These large collections have forced execution efficiency issues to the forefront, and provide an opportunity for a more rigorous evaluation of these issues. I will use these large collections to evaluate my approach, such that a further contribution of my work will be an evaluation of execution efficiency issues using large, realistic document collections.

In summary, my contributions will include the following:

- adaptation of known IR query optimization techniques to the inference network retrieval model as well as development of new optimization techniques
- an architecture demonstrating the effectiveness of using an extensible data management package to support the data management requirements of an IR system and *enable* the query optimization techniques described above
- an empirical evaluation of query optimization techniques in the inference network model using large, realistic document collections

While my results will be specific to the inference network model, they will have relevance to other retrieval models. My discussion and evaluation of query optimization techniques will provide information on the performance of individual techniques that are applicable to other models, and also describe how

they may be effectively combined. The ability to integrate different retrieval strategies is not unique to the inference network model, so any result that demonstrates how an integrated framework of retrieval strategies can be optimized will be interesting. Furthermore, my implementation of the inverted file will be relevant to any system that uses an inverted file to index the document collection.

In the next section I provide background information on the two systems that will be used in this study. In Section 3 I describe the characteristics of an inverted file that must be considered when integrating the two systems and the results of a preliminary integration. In Section 4 I discuss known IR query optimization techniques, propose new optimization techniques, and describe how these techniques will be evaluated. Section 5 contains a discussion of related work, and Section 6 offers some concluding remarks.

2 Background

My proposed research will be conducted using two existing systems: the INQUERY full-text information retrieval system, and the Mneme persistent object store. In the following subsections I will give an overview of each of these systems.

2.1 INQUERY

INQUERY is a full-text information retrieval system based on a Bayesian inference network model. The inference network retrieval model is rooted in the probabilistic retrieval model. I will begin my discussion of INQUERY with a brief review of the probabilistic retrieval model. This review moves quickly, so fasten your seat-belt. Following that, I will give a more detailed discussion of the Bayesian inference network model as implemented by INQUERY.

2.1.1 Probabilistic Retrieval

The probabilistic retrieval model was first suggested in 1960 by Maron and Kuhns [35]. The basic idea is to rank the documents in a collection based on their probability of being relevant to the current information need. This is expressed as $P(relevant \mid d)$, or the probability that the information need is met given document d. A user's information need is something internal to the user and cannot be expressed exactly to the system, so this probability must be estimated using the terms supplied by the user in a query. The estimation is simplified using a version of Bayes' theorem to rewrite the probability as

$$P(relevant \mid d) = \frac{P(d \mid relevant)P(relevant)}{P(d)}$$

Document d can be represented as a binary vector $\mathbf{x} = (x_1, x_2, \ldots, x_v)$, where $x_i = 1$ if term i appears in document d, $x_i = 0$ otherwise, and the terms are (typically) limited to those that appear in the query. Now the estimation task amounts to estimating the probability of the terms appearing in a relevant document, $P(\mathbf{x} \mid relevant)$, and the *a priori* probability of a document, $P(\mathbf{x})$. $P(relevant)$ will be constant for a given query and so may be ignored.

Robertson and Sparck Jones [45] revised the probabilistic model into its current form. They observed that a document should be retrieved if its probability of being relevant is greater than its probability of being not relevant, $P(relevant \mid d) > P(not\ relevant \mid d)$. For the purposes of ranking the documents in a collection, this can be restated as a cost function

$$g(\mathbf{x}) = \log \frac{P(\mathbf{x} \mid relevant)}{P(\mathbf{x} \mid not\ relevant)} + \log \frac{P(relevant)}{P(not\ relevant)}$$

where document d is expressed as the binary vector $\mathbf{x}$, Bayes' theorem has been used, and the logs have been introduced to linearize the function.

If we assume that terms appear independently in the relevant documents, we can rewrite $P(\mathbf{x} \mid relevant)$ as $P(x_1 \mid relevant)P(x_2 \mid relevant) \cdots P(x_v \mid relevant)$, and similarly for the not relevant case. Let $p_i = P(x_i = 1 \mid relevant)$ and $q_i = P(x_i = 1 \mid not\ relevant)$, then

$$P(\mathbf{x} \mid relevant) = \prod_i p_i^{x_i}(1 - p_i)^{(1-x_i)}$$

and

$$P(\mathbf{x} \mid not\ relevant) = \prod_i q_i^{x_i}(1 - q_i)^{(1-x_i)}$$

Our cost function can now be rewritten as

$$g(\mathbf{x}) = \sum_i x_i \log \frac{p_i(1 - q_i)}{(1 - p_i)q_i} + \sum_i \log \frac{1 - p_i}{1 - q_i} + \frac{P(relevant)}{P(not\ relevant)}$$

The last two terms will be constant for a given query (since x_i does not appear in them), so we are left with the first term as our ranking function. This is known as the *binary independence* model.

We are still faced with the problem of estimating p_i and q_i. The solution is to use some other technique to return an initial set of documents to the user and obtain feedback about the relevant and non-relevant documents in the set. The distribution of query terms in the relevant and non-relevant documents in this sample is then used to estimate p_i and q_i, and the query is re-evaluated probabilistically. Croft and Harper [16] showed how the probabilistic model could also be used for the initial search. They assume that p_i is the same for all terms and q_i can be estimated with n_i/N, where n_i is the number of documents in which term i occurs and N is the number of documents in the collection. The ranking function now becomes

$$g(\mathbf{x}) = C \sum_i x_i + \sum_i x_i \log \frac{N - n_i}{n_i} \tag{1}$$

This is referred to as the *combination match*, which applies the constant factor C times the number of matches between the terms in the query and the terms in the document, plus what is essentially the inverse document frequency of each query term that appears in the document.

Equation 1 assumes that a term is either fully assigned to a document, or not at all. The mere appearance of a term in a document, however, does not necessarily mean that the term is indicative of the contents of the document. Rather than make such extreme judgments, we would prefer to use a finer granularity when expressing the degree to which a term should be assigned to a document. This was accomplished by Croft [14, 15] who expressed this degree as the probability of a term being assigned to a document, $P(x_i = 1 \mid d)$, such that documents should now be ranked by the *expected* value of Equation 1, or

$$g(\mathbf{x}) = \sum_i \left[P(x_i = 1 \mid d) \left(C + \log \frac{N - n_i}{n_i} \right) \right]$$

$P(x_i = 1 \mid d)$ is then estimated using the normalized within document frequency of the term, $ntf_{id} = tf_{id}/max\_tf_d$, where tf_{id} is the number of occurrences of term i in document d, and max_tf_d is the maximum of $\{tf_{1d}, tf_{2d}, \ldots\}$. To increase the significance of even a single occurrence of a term in a document, a constant K in the range 0 to 1 is applied to yield the final probabilistic ranking function

$$g(\mathbf{x}) = \sum_i \left[(K + (1 - K)ntf_{id}) \left(C + \log \frac{N - n_i}{n_i} \right) \right] \tag{2}$$

2.1.2 Bayesian Inference Network Retrieval

The Bayesian inference network model generalizes the probabilistic retrieval model by treating retrieval as an evidential reasoning process where documents are used as evidence to estimate the probability that a user's information need is met. An inference network consists of nodes and directed edges between the nodes forming a directed acyclic graph (DAG). The nodes represent binary valued (i.e., true or false) propositional variables or constants and the edges represent dependencies between the nodes. If the proposition represented by a given node p implies the proposition represented by node q, then a directed edge is drawn from p to q. Node q will also contain a *link matrix* that specifies the probability of q given p, $P(q \mid p)$, for all possible values of p and q. Since p and q may each be either true or false, this link matrix will contain four entries. If q has multiple parents (π_q), the link matrix will specify the conditional probability of q on the set of parents, $P(q \mid \pi_q)$. Typically the network is large such that storing the entire link matrix for a node is impractical. Instead, the link matrix is represented in a canonical form and we store only the information required to compute each matrix entry from the canonical form.

If the probabilities of the root nodes in the network are known, Bayesian inference rules can be used to condition these probabilities over the rest of the network and compute a probability, or belief, for each of the remaining nodes in the network. Moreover, if our belief in any given proposition should change, its probability can be adjusted and the network can be used to update the probabilities at the rest of the nodes.

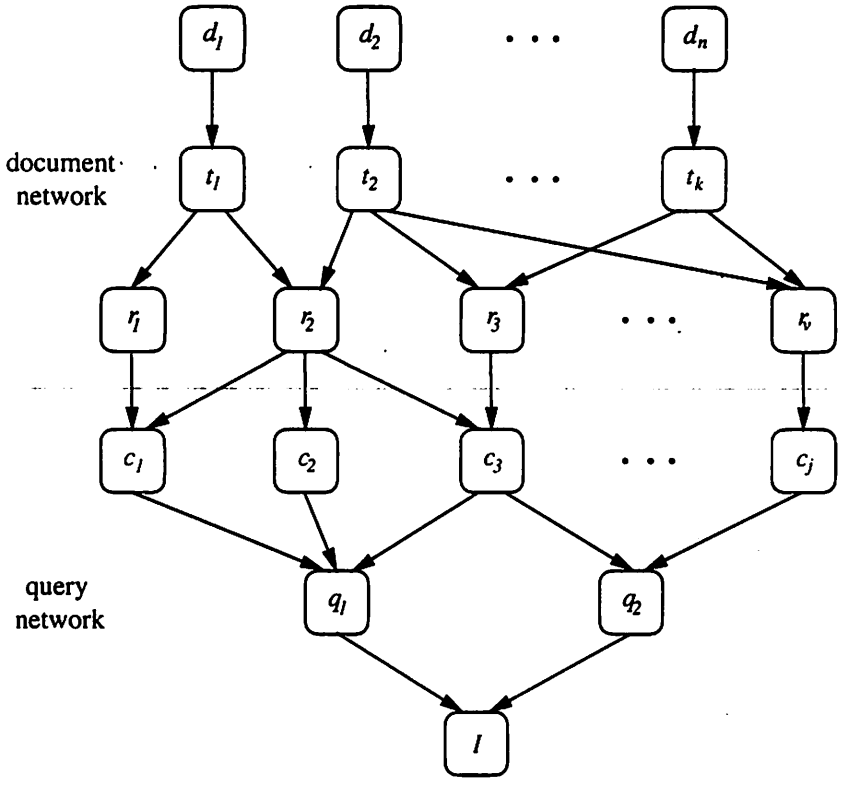

Figure 1: Inference network for information retrieval.

The application of Baysien inference networks to information retrieval was advanced by Turtle and Croft [54, 56, 55]. The inference network used for information retrieval is divided into two parts, a document network and a query network, shown in Figure 1. The document network consists of document nodes (d_i's), text representation nodes (t_i's), and concept representation nodes (r_i's). A document node

represents the event that a document has been observed at an abstract level, while a text node represents the event that the actual physical content of a document has been observed. This distinction is made to support complex documents which may have multiple physical representations (e.g., multimedia documents with text and video), and sharing of the same physical text by multiple documents (e.g., if two documents are merely different published forms of the same text). In the first case, a document node will have multiple children text nodes, while in the second case, a text node will have multiple parent document nodes. Typically, each document has only one text representation and the text representations are not shared by multiple documents, such that the document network may be simplified by eliminating the text nodes.

A concept representation node represents the event that a document concept has been observed. Document concepts are the basic concepts identified in the document collection. Commonly these are the terms in the document collection, but they may also be more semantically meaningful concepts extracted from the text by sophisticated indexing methods. The conditional probability $P(r_i \mid d_j)$ stored in a concept representation node quantifies our estimate of the degree to which the concept should be assigned to the document, as well as the ability of the concept to describe the information content of the document. This estimate can be borrowed from the probabilistic retrieval model, using Equation 2 as the foundation of the estimate.

The query network consists of query concept nodes (c_i's), query nodes (q_i's), and a single information need node (I). Node I represents the event that a user's information need has been met. Query nodes are a representational convenience that allow the information need to be expressed in multiple query forms. They represent the events that particular query forms have been satisfied, and could be eliminated by using more complicated conditional probabilities at node I. Query concepts are the basic concepts used to represent the information need. A query concept node describes the mapping between the concepts used in the document representation and the concepts used in the query representation, and will have one or more document concept representation nodes for parents. In the common case, each query concept node will have a single parent.

The document network is constructed once at indexing time. The links between the nodes and the link matrices stored within the nodes never change. The query network is constructed when the query is parsed. The link matrix stored in a query node will be based on the query operator represented by the node. Such operators might include the boolean operators, simple sums, or weighted sums where certain query concepts have been identified as being more significant and consequently given more weight. The link matrix in the information need node will describe how to combine the results from the different query representations. Unlike the document network, the conditional probabilities in the query network may be updated given additional information from the user, as might occur during relevance feedback.

The inference network is used by attaching the roots of the query network to the leaves of the document network. To produce a score for document d_j, we assert $d_j = true$ and $d_k = false$ for all $k \neq j$, and condition the probabilities through the network to obtain $P(I \mid d_j)$. If a document provides no support for a concept (i.e., it doesn't contain that term), a default belief is assigned to that concept node when conditioning over the network. A score is computed in this way for all documents in the collection, which are then ranked based on their scores. In practice, we need only compute scores for documents which contain at least one of the query concepts. As the query is evaluated, a default document score is computed which is then assigned to all documents which contain none of the query terms.

Given that retrieval is driven by evaluating the inference network for the documents that contain the query terms, the logical choice of index to support this process is an inverted file [46, 18, 26]. An inverted file index consists of a record, or inverted list, for each term that appears in the document collection. A term's inverted list stores a document identifier and weight for every document in which the term appears. Generally speaking, the weight might simply be the number of times the term appears in the document, or it might be a more sophisticated measure of the significance of the term's appearance in the document. Additionally, the location of each occurrence of the term in the document may be stored in order to support proximity operators, or queries based on the relative positions of terms within documents.

In the case of INQUERY, proximity operators are supported so the location of each occurrence of a term is stored in the inverted lists. The weight for document d_j that is stored in the inverted list for term r_i is simply the within document frequency, tf_{ij}, rather than the belief, $P(r_i \mid d_j)$. There are two reasons for this choice. First, INQUERY is a research system and new term weighting schemes are constantly being tested. By computing the belief at a document representation node during retrieval rather than storing it directly in the inverted list, new weighting schemes can be tested without re-indexing the document collection. Second, a belief involves per document and per term constant factors for normalization. These will change as new documents are added to the collection or existing documents are modified, such that any stored beliefs would have to be updated. Moreover, storing the beliefs would amount to redundantly storing these constant factors in every document weight, greatly reducing the compressibility of these weights, as compared to storing within document frequencies alone.

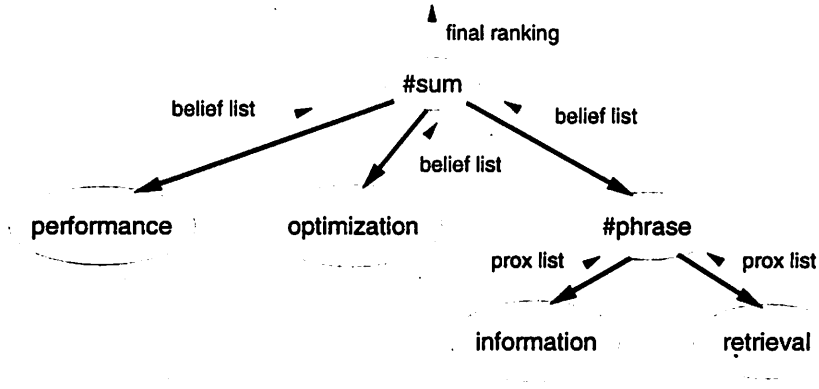

Figure 2: Example query in internal tree form.

Using these inverted lists, retrieval is actually implemented as follows. First, the query is represented internally as a tree with query operators at the internal nodes and terms at the leaves. This is essentially the query network of Figure 1 with the direction of the arcs reversed. An example is shown in Figure 2, where query operators are prefixed with a hash mark (#) and the arcs are annotated with the type of information flowing back from the child. Each node in the tree is evaluated in depth-first order. A term (leaf) node is evaluated by retrieving the inverted list for the term and returning either a *belief list* or a *proximity list*, depending on what is expected by the parent. A belief list contains the computed beliefs for all documents that contain the term, and would be expected by an operator that manipulates beliefs, such as a boolean or sum operator. A proximity list contains the locations of all occurrences of the term and would be expected by a proximity operator, such as a phrase operator. An operator (internal) node is evaluated according to the semantics of the operator and returns the type of list expected by its parent, with the proviso that a belief list can always be obtained from a proximity list, but a proximity list can never be obtained from a belief list (the query language grammar precludes nesting a belief operator inside a proximity operator, so once a belief list is created, the location information is discarded). The belief list returned by the operator at the root of the tree is sorted to produce the final ranked output.

The above scheme is referred to as "term-at-a-time" processing, where the beliefs for all documents are updated simultaneously by a given operator. The current implementation is actually a variation of this scheme, called "document-at-a-time" processing, where the complete final score for a single document is computed before moving on to the next document. Processing begins by selecting the smallest document identifier from all of the inverted lists involved in the query and evaluating the entire query for just this document. Identifying the smallest document identifier is straight forward because the inverted list entries

are sorted by document identifier. The next smallest document identifier is then selected and the query is evaluated for that document. This process continues until all documents in the inverted lists for the terms in the query have been processed.

Term-at-a-time processing produces large intermediate results. When an operator is evaluated, an intermediate result list must be allocated to accumulate the scores returned by each of the operator's children as they are evaluated. For example, in a collection with 1 million documents, a single belief list would consume nearly 8 Mbytes (each entry has a 4 byte document identifier and a 4 byte belief value). Given the depth-first evaluation order, the number of intermediate result lists that might be needed simultaneously is equal to the depth of the query tree. On platforms that do not have enough main memory to accommodate these intermediate lists, the lists must be written to disk.

With document-at-a-time processing, a full belief list is required only at the root. The internal nodes require just a single belief or proximity entry for the current document being evaluated, eliminating the large intermediate result lists and the need to save these lists to disk. Avoiding the reading and writing of intermediate result lists improves the overall execution time.

Note, however, that document-at-a-time processing may have prohibitively high main memory costs as well. Each term involved in the query must have a main memory buffer to hold the portion of the term's inverted list that contains information for the document currently being evaluated. Large queries with many terms may exceed the main memory capacities of some platforms, such that term-at-a-time processing with intermediate results saved to disk is the only feasible processing technique. In the case where there is sufficient main memory to hold the intermediate lists of term-at-a-time processing, then term-at-a-time processing will most likely be less expensive than document-at-a-time processing since the later must evaluate the query tree multiple times, once for each document that appears in the inverted lists for the terms in the query.

2.2 Mneme

The Mneme persistent object store [40] was designed to be efficient and extensible. The basic services provided by Mneme are storage and retrieval of objects, where an object is a chunk of contiguous bytes that has been assigned a unique identifier. Mneme has no notion of type or class for objects. The only structure Mneme is aware of is that objects may contain the identifiers of other objects, resulting in inter-object references.

Objects are grouped into files supported by the operating system. An object's identifier is unique only within the object's file. Multiple files may be open simultaneously, however, so object identifiers are mapped to globally unique identifiers when the objects are accessed. This allows a potentially unlimited number of objects to be created by allocating a new file when the previous file's object identifiers have been exhausted. The number of objects that may be accessed *simultaneously* is bounded by the number of globally unique identifiers (currently 2^{28}).

Objects are physically grouped into *physical segments* within a file. A physical segment is the unit of transfer between disk and main memory and is of arbitrary size. Objects are also logically grouped into *pools*, where a pool defines a number of management policies for the objects contained in the pool, such as how large the physical segments are, how the objects are laid out in a physical segment, how objects are located within a file, and how objects are created. Note that physical segments are not shared between pools. Pools are also required to locate for Mneme any identifiers stored in the objects managed by the pool. This would be necessary, for instance, during garbage collection of the persistent store. Since the pool provides the interface between Mneme and the contents of an object, object format is determined by the pool, allowing objects to be stored in the format required by the application that uses the objects (modulo any translation that may be required for persistent storage, such as conversion of main memory pointers to object identifiers). Pools provide the primary extensibility mechanism in Mneme. By implementing new

pool routines, the system can be significantly customized.

The base system provides a number of fundamental mechanisms and tools for building pool routines, including a suite of standard pool routines for file and auxiliary table management. Object lookup is facilitated by *logical segments*, which contain 255 objects logically grouped together to assist in identification, indexing, and location. A hash table is provided that takes an object identifier and efficiently determines if the object is resident in main memory. Support for sophisticated buffer management is provided by an extensible buffering mechanism. Buffers may be defined by supplying a number of standard buffer operations (e.g., allocate and free) in a system defined format. How these operations are implemented determines the policies used to manage the buffer. A pool *attaches* to a buffer in order to make use of the buffer. Mneme then maps the standard buffer operation calls made by the pool to the specific routines supplied by the attached buffer. Additionally, the pool is required to provide a number of "call-back" routines, such as a modified segment save routine, which may be called by a buffer routine.

Mneme is particularly appropriate for the task of managing an inverted file for a number of reasons. First, an object store provides the ideal level of functionality and semantics. The data that must be managed can be naturally decomposed into objects, where each inverted list is a single object. More sophisticated mappings of inverted lists to objects can also be easily supported with inter-object references, which allow more complex data structures to be built up. The primary function required is object retrieval, or providing access to the contents of a given object for higher level processing. Object access includes the traditional data management tasks of buffering and saving modifications. The processing of objects, however, is highly stylized and unlikely to be adequately supported within the data management system. Therefore, semantic knowledge about the contents of an object within the data management system is not only useless, but actually cumbersome. An object store that treats objects as containers of uninterpreted bytes and inter-object references provides just the right level of semantics.

Second, because Mneme is extensible, certain functions can be customized to better meet the management requirements of an inverted file. The objects in an inverted file will come in a variety of sizes and exhibit unusual access patterns, such that a single physical storage scheme specifying clustering and physical segment layout will be inadequate. A better approach will be to identify groups of objects that can benefit from storage schemes tailored to the physical characteristics and access patterns of each group. In particular, buffer management policies should be customized for each group.

Finally, Mneme is tuned for performance and imposes a particularly low overhead along the critical path of object access. Resident objects are quickly located using the resident object table, and non-resident objects are faulted in with little additional processing. This can be contrasted with page mapping architectures of other object stores [28, 50] which have a fairly high penalty for accessing a non-resident object. These systems are optimized for localized processing of a large number of small objects, where the cost of faulting a page of objects can be amortized over many access to the objects in the page. This pattern of access differs from that expected in an inverted file, where large objects are accessed for sequential processing with little temporal locality.

3 System Integration

The goal of the first phase of this research was to demonstrate the feasibility of using the Mneme persistent object store to manage INQUERY's inverted file. This goal has been satisfactorily achieved. The initial integration was straight forward and will provide a base line for comparison in the next phase when query optimizations are considered. The effectiveness of the integration depended on the extent to which Mneme's extensibility features could be used to customize the inverted file management based on the size and access characteristics of the objects in the inverted file. In the next subsection I briefly examine these characteristics, and following that I describe the integrated system and its performance.

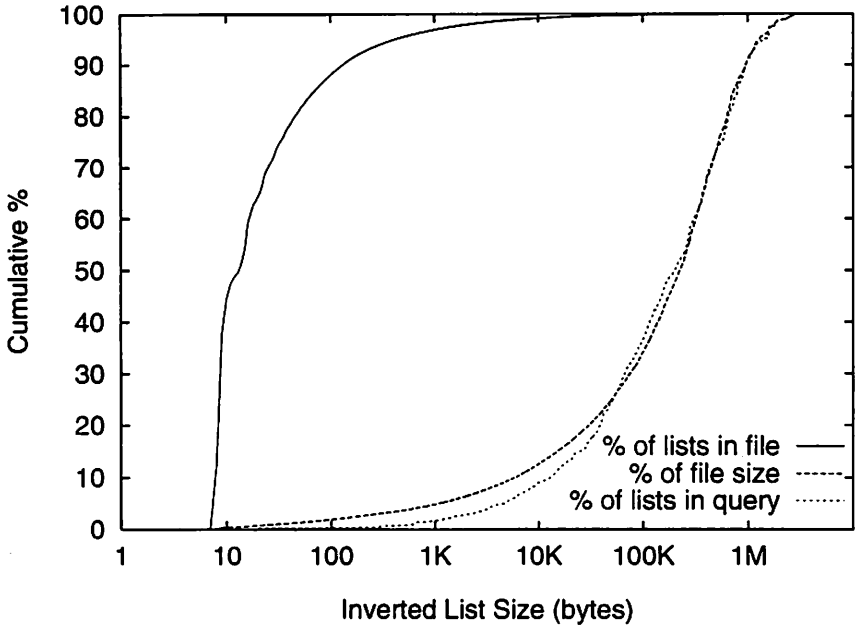

Figure 3: Cumulative distributions over inverted list size for TIPSTER collection, with 846331 lists and 720 Mb total.

3.1 Inverted List Characteristics

The size of an inverted list depends on the number of occurrences of the corresponding term in the document collection. Zipf [62] observed that if the terms in a document collection are ranked by decreasing number of occurrences (i.e., starting with the term that occurs most frequently), there is a constant for the collection that is approximately equal to the product of any given term's frequency and rank order number. The implication of this is that most of the terms will occur a relatively small number of times, while a few terms will occur very many times.

Figure 3 shows the distribution of inverted list sizes for the 2 Gbyte TIPSTER document collection [24]. For a given inverted list size, the figure shows how many records in the inverted file are less than or equal to that size, and how much those records contribute to the total file size. As we would expect, the majority of the inverted lists are relatively small—approximately 95% of the lists are less than 1 Kbyte. In fact, better than 50% of the lists are less than 16 bytes. It is also clear that these small lists contribute a very small amount to the total file size. Less than 5% of the total file size is accounted for by inverted lists smaller than 1 Kbyte. Put another way, better than 95% of the total file size is accounted for by less than 5% of the inverted lists in the file. Obviously, the lists in this 5% can be quite large, with the largest list in the file weighing in at 2.5 Mbytes.

If we could assume that inverted list access during query processing was uniformly distributed over the inverted lists, then supporting this activity (from a data management perspective) would be simplified, since the majority of the file accesses would be restricted to a relatively small percentage of the overall file. Unfortunately, this is not the case. Figure 3 also shows the distribution of sizes for the inverted lists accessed by a typical query set (produced from *TIPSTER Topics 51–100*). The majority of the records accessed are between 10 Kbytes and 1 Mbyte. This size range represents a small percentage of the total number of records in the file, but a large percentage of the total file size. Therefore, we must be prepared to provide efficient access to the majority of the raw data in the file.

We can, however, anticipate one access characteristic during query processing that works in our favor. It is likely that there will be non-trivial repetition of the terms used from query to query. This can be expected for two reasons. First, a user of an IR system may iteratively refine a query to obtain the desired set of documents. As the query is refined to more precisely represent the user's information need, terms from earlier queries will reappear in later queries. Second, IR systems are often used on specialized collections where every document is related to a particular subject. In this case, there will be terms that are common to a large number of queries, even across multiple users. The upshot of this is that caching inverted lists in main memory should prove beneficial.

3.2 The Integrated Architecture

The initial integration is fully discussed and evaluated in [7]. Here I merely summarize the architecture and highlight its performance. The Mneme version of the inverted index was created by allocating a Mneme object for each inverted list in the inverted file. The inverted lists were divided into three distinct groups. First, more than 50% of the inverted lists are 12 bytes or less. By allocating a 16 byte object (4 bytes for a size field) for every inverted list less than or equal to 12 bytes, we can conveniently fit a whole logical segment (255 objects) in one 4 Kbyte physical segment. This simplifies both the indexing strategy used to locate these objects in the file and the buffer management strategy for these segments. Inverted lists in this category were allocated in a *small object pool.* Second, a number of inverted lists are so large it is not reasonable to cluster them with other objects in the same physical segment. Instead, these lists are allocated in their own physical segment. All inverted lists larger than 4 Kbytes were allocated in this fashion in a *large object pool.* The remaining inverted lists form the third group and were allocated in a *medium object pool.* These objects are packed into 8 Kbyte physical segments. The physical segment size is based on the disk I/O block size and a desire to keep the segments relatively small so as to reduce the number of unused objects retrieved with each segment.

This partitioning of the objects allows the indexing and buffer management strategies for each group to be customized. Each object pool was attached to a separate buffer, allowing the global buffer space to be divided between the object pools based on expected access patterns and memory requirements. The buffer replacement policy for all of the pools is least recently used (LRU) with a slight optimization. Before the query tree built by INQUERY is processed, it is scanned and any objects required by the query that are already resident are "reserved", potentially avoiding a bad replacement choice.

The performance of the integrated system was evaluated by comparing its query processing speed with that of the original version of INQUERY, which used a custom b-tree implementation of the inverted file. A variety of document collections were used in the evaluation, ranging in size from 2 Mbytes to 2 Gbytes. The Mneme version of INQUERY required 6% to 29% less wall clock time to process queries than the b-tree version of INQUERY. The reduction in query processing time was due solely to reductions in the cost of obtaining data from the inverted file. This cost was isolated, and it was found that the Mneme version reduced this cost by 25% to 64%.

The performance improvement enjoyed by the Mneme version of the system can be attributed to careful file allocation sympathetic to the device transfer block size and intelligent caching of auxiliary tables and inverted lists. The results demonstrate the feasibility of using an "off-the-shelf" data management package to support the data management requirements of an IR system. They also indicate the importance of considering lower level disk and I/O issues when addressing the execution performance of an IR system. Having established the feasibility of the proposed architecture, I will now move on to consider higher level query optimization issues.

4 Query Optimization

My pursuit of query optimization techniques will be guided by an overall strategy of maximizing return on I/O. This strategy has a number of attractive features. First, it attempts to reduce the most expensive operation in the system—retrieving data from disk. Disk reads are six orders of magnitude slower than main memory reads [22]. Reducing the amount of I/O that must be performed should produce a significant reduction in execution time. Second, a reduction in I/O during query processing will inevitably lead to a reduction in computation (in the absence of compression). Data that no longer needs to be read in obviously no longer needs to be processed, providing further, although less significant, execution savings. Finally, this strategy incorporates the supposition that some (possibly much) of the data read in under current query processing strategies does not contribute significantly to resolving the query. This supposition is based on the function used to rank documents, combined with the nature of the terms that appear in user queries. Let us consider for a moment simple sum queries, where each query term's weight in a document is averaged to compute the document's relevance score. A component of the term weighting function is the term's inverse document frequency in the collection. The effect of this component is that terms that appear more frequently in the collection will contribute less to a given document's relevance score. More frequent terms by definition have longer inverted lists, requiring more I/O in order to be processed. The ensuing situation is that the most expensive terms to process contribute the least to the final result. Moreover, relatively frequent terms appear quite often in user queries, such that there is great potential for savings.

The space of techniques to maximize ROIO can be divided into two categories, safe and unsafe. Safe techniques have no impact on the retrieval effectiveness of the system, while unsafe techniques may trade off retrieval effectiveness for execution efficiency. Of course, the techniques that may actually be applied in INQUERY depend on the query processing operations supported by the system. I will begin my discussion of query optimization techniques by briefly reviewing INQUERY's query operators. Next, I will discuss safe techniques that are immediately relevant to INQUERY. Following that, I will discuss unsafe techniques, first at a generic level applicable to many ranking retrieval systems, and then more specifically with respect to the inference network model. In this discussion I will introduce a processing strategy that allows the incorporation of unsafe techniques into the inference network model, representing a central component of the research proposed here. As each optimization technique is discussed, I will identify the functionality requirements imposed by the technique on the inverted list implementation. After all of the techniques have been discussed, I will describe an inverted list implementation that supports the optimization techniques applicable to the inference network model. Finally, I will describe my plan to evaluate these techniques.

4.1 Query Operators

INQUERY supports a variety of query operators at the user interface level. As mentioned briefly in Section 2.1, during query processing these operators are assigned an internal evaluation function and assembled into a query tree with operators at the internal nodes and terms at the leaves. The internal evaluation functions include: **and, or, not, sum, weighted sum, maximum, ordered distance n, unordered window n, synonym**, and **passage sum**. The first six functions operate on belief values, while the last four functions operate on proximity values.

The first three functions are probabilistic implementations of the boolean operators. In traditional boolean systems, the documents in the answer set must exactly match the boolean query formula [46]. In the probabilistic model, the assignment of a term to a document is expressed with a probability or belief. A boolean formula of terms will therefore have some probability of matching a given document, allowing documents to be ranked by this probability. One implication of this is that a conjunction of terms *cannot* automatically eliminate a document from consideration just because one of the terms is absent from the document. A term that does not appear in a document will still have a default probability of representing

that document. A boolean operator node in the query tree is evaluated by logically combining the beliefs from the child nodes. If a child node does not directly compute a belief for a given document (e.g., the child node is a term that does not appear in the document), the child's default belief is used.

The **sum** and **weighted sum** functions return the average and weighted average, respectively, of the beliefs from the children in the query tree. The **sum** function provides the most general combination of belief in the query tree, and could be used, for example, to generate a document ranking from a simple list of query terms. The **weighted sum** function enhances this operation by associating a weight with each of the children in the query tree, allowing relative importance to be indicated amongst the children. As with the boolean operators, if a child node does not directly compute a belief for a given document, the child's default belief is used.

The last belief operator, the **maximum** function, returns the maximum of the beliefs from its children. In this case, if a child node does not calculate a belief for the document in question, it does not participate in the maximum and default belief values are ignored.

The next two functions are generically called proximity operations. The **ordered distance n** function identifies documents that contain all of the concepts $\{c_1 \ldots c_k\}$ represented by the child nodes in the query tree. 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. This calculation is accomplished using proximity lists, where a proximity list contains the identifiers of all of the documents that contain the concept, as well as the location of each occurrence of the concept within those documents. If any of the concepts in the proximity do not appear in a given document or do not satisfy the positional constraints within the document, the proximity "fails" for that document. In this sense, a proximity is similar to a traditional boolean conjunction. The **unordered window n** function similarly identifies documents that contain all of the concepts represented by the child nodes in the query. However, all of the child concepts must appear within a window of size n, and they may appear in any order.

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. In effect, the concepts represented by the child nodes of a **synonym** function are treated as synonyms, and the new proximity list created represents a new concept that occurs anywhere any of the child concepts occur.

The last function, **passage sum**, calculates a belief for a document in the following fashion. First, the document is divided into fixed size overlapping passages, where the last half of each passage overlaps the first half of the next passage. Then, 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, while a belief list is returned from the passage node.

A leaf of the query tree, or a term, is evaluated by retrieving the inverted list for the term. Proximity information can be obtained directly from the inverted list. The belief contribution of term i to document j is calculated with the following formula:

$$\text{belief}_{ij} = C + (1 - C)\,\text{ntf}_{ij}\,\text{nidf}_i \tag{3}$$

where

$$\text{ntf}_{ij} = Ks + (1 - K)\left(\frac{\log(\text{tf}_{ij} + 0.5)}{\log(\text{max\_tf}_j + 1.0)}\right)$$

$$\text{nidf}_i = \frac{\log((N + 0.5)/n_i)}{\log(N + 1.0)}$$

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 number of documents in the collection
n_i is the number of documents in which term i appears

The constants C and K may be specified by the user. C defines the baseline, or default, belief value and has a system default value of 0.4. K serves the same purpose as in Equation 2, acting to increase the significance of even a single occurrence of a term in a document. Its system default value is typically 0.4. s is used 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.

The belief contribution of a more complex concept, such as might be represented by a proximity node in the query tree, is calculated in the same way. The belief calculation used for a passage in the **passage sum** operator is a slightly modified version of this equation.

4.2 Safe Techniques

The simplest way to improve ROIO is to identify any data that is read in but not used, and eliminate the unnecessary I/O. One situation where this occurs results from the current structure of the inverted lists. Recall that an inverted list contains both term weights and location information. However, from the previous subsection we can see that the query operators require either just the weight information or just the location information, such that some information will be read in but not used. Therefore, it might be profitable to separate out the location information from the term weights and only read in the information actually required by the query operator. This rather obvious solution is dependent on the ability to restrict the read of an inverted list to specific portions of the list, an ability that is in fact central to most of the techniques that I will consider for maximizing ROIO, but not supported by typical inverted file implementations. In this case, the specific requirement is the ability to access just the weights for a term, just the location information for a term, or both.

The next safe technique can generally be called an intersection optimization, and is borrowed from the boolean retrieval model. In that model, a query consisting of a conjunction of terms can be evaluated in the following fashion. First, a candidate document set is created consisting of the set of documents in which one of the terms appears. Then, for each of the remaining terms, the set of documents in which that term appears is intersected with the set of candidate documents. After all terms have been processed, the candidate document set will consist of the documents which satisfy the conjunction. This process can be improved by starting with the term that appears in the smallest number of documents and processing the remaining terms in increasing order of document frequency. At each intersection, it is only necessary to check if the current term appears in the documents in the candidate set, since the candidate set can only shrink or stay the same. Therefore, savings can be realized if we can access just the portions of the inverted list for the current term that might contain an entry for a candidate document. Furthermore, if the candidate set should become empty, processing can stop immediately.

Unfortunately, the conjunction operation in the probabilistic retrieval model is not a strict intersection, so this optimization is not applicable to the **and** operator. However, the proximity operations described above *are* strict intersections in the sense that every term in a proximity must appear in a document (and satisfy any ordering and window constraints) in order for the document to satisfy the proximity. Therefore, the exact same technique can be used to improve ROIO for proximity operations. The functionality required is the ability to access just that portion of an inverted list that might contain an entry for a given document, where "portion" is the amount read in a single disk read.

The final safe technique for improving ROIO is inverted list compression. Assume that we have u bytes of data which can be compressed down to z bytes, $z < u$. If the cost of decompressing z bytes of data is less than the cost of reading $u - z$ bytes from disk, then ROIO has potentially been increased. Note also that if the cost of decompressing z bytes exceeds the cost of reading the $u - z$ extra bytes in an uncompressed inverted list, then ROIO will decrease. The increase in ROIO in the first case is *potential* because the extra inverted list information, E, that has been read in "for free" must in fact contribute towards resolving the query. If E does not contribute in this way, the ROIO could actually decrease if, for example, other currently resident inverted list data that must yet be accessed to resolve the query is flushed from main memory to make room for the decompressed E. In the second case above where the decompression is more expensive than the read, ROIO will always suffer, regardless of the contribution of E, since it would have been cheaper to read in an uncompressed version.

Compression techniques for inverted lists have received a fair amount of attention [58, 36, 30, 1, 5, 63]. I do not claim anything novel with respect to compression. Rather, for completeness I merely describe how it fits into my optimization strategy and give a necessary condition for providing benefit with respect to execution performance. I also note that compression clearly has other desirable side effects, e.g. reduced disk space requirements, whose benefits may outweigh any execution performance impacts.

4.3 Unsafe Techniques

The remaining techniques that I will consider for improving ROIO are unsafe in the sense that execution efficiency may improve at the expense of retrieval effectiveness. The unsafe techniques are based on the supposition that some of the data read in during query processing does not contribute significantly to resolving the query. The challenge is to identify this data so that we can avoid reading it in. I will classify a number of techniques to accomplish this and describe them at a generic level. I will then discuss how they might be adapted for the inference network model.

Consider a query consisting of a single belief operator and terms. Each term contributes a belief for every document in which it appears. To evaluate the query, the beliefs for a given document are combined according to the semantics of the operator to produce a final score for the document. The documents are then ranked by their final scores to produce the answer to the query. In essence, this procedure involves allocating an array large enough to hold an identifier and final score for each document, updating this array as each belief value from the terms is processed, and sorting the final array by score.

Our goal is to avoid processing the belief values that do not contribute significantly to the final document ranking. This can be accomplished by identifying some subset of the belief values that will result in a final ranking close to the "exact" ranking achieved when all belief values are processed. As this subset becomes smaller and smaller, we expect the final ranking to differ more and more from the exact ranking. The question now is how to select this subset. There are a variety of methods to make this selection, and they all can be classified based on how they decide the following: which belief value to process next, and when to stop. Both of these seemingly simple questions have interesting and subtle implications for performance and implementation. The order in which belief values are processed will affect the rate at which the array of scores is populated with discriminating information, and has implications for the inverted list organization. The stopping condition is intimately related to the belief processing order and will determine how much work will be done to answer the query and what claims can be made about the quality of the answer returned.

4.3.1 Belief Value Magnitude Ordering

The first belief value processing order we might consider is to greedily process belief values in order of decreasing contribution to the final ranking. If we consider just the **sum** and **weighted sum** operators, this is equivalent to processing belief values in order of decreasing magnitude and weighted magnitude,

respectively. This ordering is very appealing in that the document ranking scores will initially grow very quickly and the relative order of the documents should be established early in the processing. Belief values processed later in the order will be smaller, having less chance to change the relative ranking of the documents.

To support this processing order, the belief values must be extracted from the inverted lists in decreasing sorted order. Practically speaking, this would be accomplished by storing the document information in the inverted lists in decreasing belief order. The next belief value to process would be chosen by examining the next belief value in each inverted list and selecting the largest of these values.

The stopping condition for this processing order can be defined in a number of ways. First, we might simply stop after processing some arbitrary percentage of the belief values, assuming that retrieval effectiveness is a logarithmic function of the number of belief values processed and execution time is a linear function of the number of belief values processed. Determining what these functions actually look like might be done experimentally or analytically. The problem with this scheme is that, short of processing all of the belief values, it gives us no guarantees on the correctness of the final ranking obtained. This scheme was proposed by Wong and Lee [61], who describe two estimation techniques for determining how many belief values must be processed to achieve a given level of retrieval effectiveness.

An alternative to this ad-hoc stopping condition would be a stopping condition that takes advantage of the organization of the belief values. Each term will contribute at most one belief value to each document being considered. If we keep track of which terms have contributed a belief value to a given document so far, we can calculate an upper bound on the final score for that document using the current belief values from each of the terms which have not contributed a belief value for that document (since a term's belief values are processed in decreasing sorted order). Moreover, we can use the current partially computed score for a document as a lower bound for that document's final score. At any given time, if a document's lower bound exceeds all other document's upper bounds, then further consideration of that document can stop and the document can be returned as the current best document. With this stopping condition, we can guarantee that the top n documents will be returned in the correct order, making the scheme safe for the top n documents. The disadvantage of this scheme is the computational costs of the required bookkeeping, which may exceed any savings in belief value processing. This scheme is described by Pfeifer et al. [44].

If we are more concerned with obtaining the top n documents and less concerned with their relative ranking, we can define another stopping condition. At any given time, an upper bound on the remaining increase in any document's score is given by the sum of the current belief values from each of the terms. Assume the documents are ranked by their current partially computed scores. When the $n + 1^{st}$ document's current score plus the upper bound on the remaining document score increase is less than the n^{th} document's score, we know that the top n documents will not change and processing can stop. We can return the top n documents, but we cannot guarantee their relative ranking.

Rather than place a hard limit on the size of the set of documents returned, thresholds can be established that determine how a belief value is processed. Such a scheme is described by Persin [42]. If a document is not in the set of documents currently being considered and has no current score (i.e., no belief values have been processed for that document), an *insertion* threshold is used to determine if a belief value for that document is significant enough to place the document into the consideration set. If the document is already in the consideration set, an *addition* threshold is used to determine if a belief value is significant enough to modify a document's current score. The addition threshold allows us to stop processing an inverted list as soon as its belief values fall below the addition threshold. The insertion threshold ensures that we consider only documents which have a significant belief contribution from the terms. With this scheme, we can make no claims about the quality of the final ranking.

4.3.2 Document Based Ordering

None of the previous schemes can guarantee that a complete score for a given document has been computed. All that might be guaranteed is that the top n documents have been returned, and in one case, that they are correctly ranked. If we require that complete final scores be calculated for all documents ranked, then belief processing order may be document driven, as in document-at-a-time query processing. In this scenario, once the current document to process has been determined, the belief values for all of the query terms that appear in the document must be processed. This requires document based access into the inverted lists and is most easily supported by storing the belief information in the inverted lists in document identifier order. Now we must decide the order in which to process the belief values for the current document. Again, the order of decreasing contribution to the document's final score is most useful. This can be approximated by processing the belief values in decreasing order of the inverse document frequency of their corresponding terms.

This per document belief value processing order allows us to use the following stopping condition. Assume we wish to return the top n documents. We begin by initializing the set of top n documents with complete scores for the first n documents, where document processing order may be defined as in Section 2.1 for document-at-a-time processing. We then identify the minimum score S from these top n documents. For each of the remaining documents, an upper bound on the current document's final score can be calculated from its currently accumulated score and the inverse document frequencies of the terms not yet processed for the document. If this upper bound becomes less than S, processing of the current document can stop because it cannot appear in the top n documents. If a complete score for the document is computed which is greater than S, the document is placed in the set of top n documents and S is recalculated. This scheme guarantees that the top n documents are returned, correctly ranked and with complete final scores. Processing savings will accrue whenever a document's upper bound descends below S and the document is eliminated from consideration before its complete score is calculated. I/O savings will accrue if we have the ability to skip portions of inverted lists. Frequent terms will occur late in the processing order and will have long inverted lists. Many documents will be eliminated from consideration before these frequent terms are processed, such that much of the inverted list information for these terms can be skipped. This scheme is called max-score by Turtle and Flood in [57].

The document processing order used above will attempt to calculate a score for every document that appears in the inverted lists of the query terms. In fact, we can identify another stopping condition at which point all document processing can stop. As processing proceeds, all of the belief values from short inverted lists will eventually be processed, such that those terms no longer need to be considered. If the upper bound contribution of the remaining terms which still have belief values to process descends below S, then all processing can stop. We may be able to achieve this condition more quickly by altering the document processing order to process first those documents which appear in the shortest inverted lists, encouraging the early exhaustion of these lists.

4.3.3 Term Based Ordering

The last belief value processing order is term based, where all of the beliefs for a given term are processed at once. This corresponds to term-at-a-time query processing. As with the per document belief value processing order above, terms are processed in decreasing order of document score contribution, approximated by the term's inverse document frequency score. This strategy will cause the terms to be processed in order of inverted list length, from shortest to longest.

The first stopping condition we will consider was originally described by Buckley and Lewit [8] and later discussed by Lucarella [31]. It is intended to eliminate processing of entire inverted lists, and is similar to the third stopping condition described in Section 4.3.1. Assume that we are to return the top n documents to

the user. 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 belief 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 we know that the set of top n documents has been found. At this point we can stop processing and guarantee that the top n documents will be returned. We cannot, however, guarantee either the relative ranking of the documents within the set or that complete scores have been calculated for those documents.

This scheme elegantly addresses the paradox stated earlier where the most expensive terms to process contribute the least to the final score. Since the terms are processed in order of decreasing score contribution, the upper bound score increase will diminish as quickly as possible, and the most expensive terms to process will be eliminated by the stopping condition. Note also that since the processing order and stopping condition are completely term based, there are no constraints on the organization of the document belief values within an inverted list.

There are three variations on this stopping condition, all of which are similar to the last stopping condition described in Section 4.3.1. The first variation was proposed by Harman and Candela [25], called *pruning*. Rather than place a limit on the number of documents returned to the user, we can establish an insertion threshold for placing new documents in the candidate set. In this case, the insertion threshold is term based, such that a term's potential score contribution must exceed some threshold in order for the term to contribute new documents to the candidate set. Processing will then have two distinct phases. First, during a disjunctive phase, documents will be added to the candidate set and partial scores updated as usual. Then, after the insertion threshold is reached, a conjunctive phase will occur where terms are not allowed to add new documents, only update the scores of existing documents. This scheme can make no guarantees about the membership of the set. It does, however, calculate complete scores for the documents in the candidate set, guaranteeing a correct relative ranking.

The second variation was proposed by Moffat and Zobel [39, 37, 38]. 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. This variation makes the same guarantees as the previous one.

The third variation is a term-at-a-time version of max-score described by Turtle and Flood [57]. New documents are added to the candidate set until the upper bound score of an unseen document (determined from the maximum possible belief score contributions of the unprocessed terms) falls below the current partial score of the n^{th} document. At this point, we know that no unseen document can appear in the top n documents. Processing then continues in a conjunctive fashion, updating the scores for just those documents currently in the candidate set. When a given document's score is updated, its maximum possible score is computed assuming it contains all of the unprocessed terms. If this maximum score is less than the n^{th} score, this document is eliminated from the candidate set. This variation will guarantee that the top n documents are returned in the correct order.

During the conjunctive processing phase of the last three variations, access into the inverted lists will be document based. This suggests that, for the most efficient processing, document belief information within the inverted lists should be sorted by document identifier. Moreover, as in Section 4.3.2, the ability to skip portions of inverted lists should provide significant I/O savings during this processing phase.

4.3.4 The Inference Network Solution

Two aspects of the unsafe techniques described above present problems when the techniques are considered for the inference network model. First, the techniques assume that document score is a monotonically

increasing function of the number of terms processed. In fact, this is true only for the **sum**, **weighted sum**, and **or** operators. Second, we have been considering only the simple case of a single belief operator and terms. The query language associated with the inference network model supports a much richer combination of operators and terms in the query tree. Even if the techniques described above were appropriate for a particular operator in the query tree, it is not clear that applying them in a localized fashion is appropriate. Most of the techniques constrain the set of documents for which scores will be returned. If this activity is not coordinated throughout the query tree, the impact on the final ranking will be unpredictable and it will be impossible to make guarantees regarding the membership and ranking of the final set of documents returned to the user. Furthermore, we may forfeit optimization opportunities by processing a document in one subtree that has already been eliminated from consideration in another subtree.

To see more clearly the impact on our confidence in the final answer, consider again the general query evaluation process in the inference network model. Evidence for a particular document's relevance to the query is gathered throughout the network and combined at the root node to produce a final belief score for the document. All of the documents can then be ranked by these final scores. If we do not gather all of the evidence for a given document, as would be the case if it were eliminated from the candidate set in some subset of the query tree, then we cannot be sure of its final score. This suggests that our optimization technique should ensure that a document's final score is always fully calculated.

To address these concerns, I offer a new processing strategy for evaluating the query network in the inference network retrieval model. Query processing begins by establishing an initial candidate set of documents, C. Starting with the root, each operator node in the query tree is evaluated in the following fashion. First, some subset of documents, C_i, is passed down from the parent node to the current node i for evaluation (in the case of the root, this subset comes from the initial candidate set). The membership of this subset is determined by the semantics of the current operator, and would typically consist of either the entire set of documents at the parent node or just a single document. Next, the children of the current node are evaluated. If a child node is an operator, this process is repeated recursively. If a child node is a term, information for the documents in C_i is extracted from the term's inverted list. When information for the documents in C_i from all of the children (inverted list information from terms, intermediate results from operators) has been assembled, the information is combined according to the semantics of the operator and passed back to the parent. If C_i was not the full set of documents at the parent, this process is repeated until the parent has passed down for evaluation all of the documents in its candidate set.

When C_i consists of all of the documents in the initial candidate set C, this processing scheme is equivalent to term-at-a-time processing. When C_i is just a single document, this processing scheme is equivalent to document-at-a-time processing. As such, the overall strategy can be considered a hybrid processing scheme. The decision at a given node as to how much of the current candidate document set to pass to a child operator will depend on the size of the current candidate set and the semantics of the child operator. If the child operator is particularly expensive to evaluate in terms of intermediate result resources, then document-at-a-time style processing should occur for that node. If it is more efficient to evaluate the entire candidate document set all at once in the child operator, then term-at-a-time processing should occur.

Processing will always be document driven and a complete score for every document in C will be calculated. If candidate document sets are processed in document identifier order, then we can ensure sequential processing of inverted lists by storing belief information in the inverted lists in document identifier order. Additionally, the ability to skip portions of long inverted lists is necessary to avoid reading in information for documents that do not appear in the candidate set.

Now all we need is some way to populate the initial candidate set. To do this, we can borrow some heuristics from the unsafe techniques above. First, infrequent terms (terms with large inverse document frequencies) are likely to make large contributions to a document's final relevance score. Therefore, they will identify good candidate documents. For all terms whose inverse document frequency exceeds some threshold, all documents that contain those terms (i.e., all documents that appear in those terms' inverted

lists) will be placed in the candidate document set. This is related to the secondary ordering heuristic used in Sections 4.3.2 and 4.3.3, where terms are processed in decreasing order of inverse document frequency. Second, more frequent terms may still contribute significant belief values for the documents in which they appear frequently. For a term that does not exceed the inverse document frequency threshold, the documents associated with the top n belief values in the term's inverted list should be added to the candidate document set. This is related to the ordering heuristic used in Section 4.3.1.

A term's inverse document frequency is inversely proportional to its inverted list length. Rather than establish an inverse document frequency threshold for candidate set population, I will 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*. All of the documents that appear in a short list will be used to populate the initial 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 is essentially free.

Populating the candidate set from the long inverted lists is more subtle. Obtaining the documents associated with the top n belief values implies that the inverted lists are sorted by document belief value. However, we have already established that the query tree evaluation is document driven and requires the inverted lists to be sorted by document identifier. If n is defined to be relatively small, then we can maintain a separate listing of the documents associated with the top n belief values for each long inverted list. Recall from Section 3.1 that there are relatively few long inverted lists, but that these lists consume the majority of the space in the inverted file. The overhead associated with top n lists will be a small percentage of this total space requirement. Exactly how n is defined remains to be determined. Some possibilities include a fixed number, say 1000, or some function of the document frequency of the term.

Population of the initial candidate set occurs during a preprocessing phase of the query tree. For the most part, we simply identify term nodes and use them to populate the initial candidate set as described above. There are some special cases. Proximity operators can be viewed as constructing new concepts (terms) at query processing time, such that they must be evaluated before we can apply our candidate set population techniques. During the preprocessing phase, a proximity operator that returns a belief list will be fully evaluated, including recursively evaluating its children. A proximity operator returns a belief list if its parent is a belief operator. Equivalently, the first proximity operator encountered on any path from the root to the leaves will always return a belief list. This belief list can then be used to populate the initial candidate set, taking either every document that appears in the list, or the documents associated with the top n belief values, where n is yet to be defined as before. The belief list is saved until the evaluation phase to avoid re-evaluating the proximity.

Another 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 the 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 a candidate document that does 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 query processing strategy proposed above offers a number of benefits. First, it provides a framework in which the fundamental concepts of the previously described unsafe techniques for constraining the candidate document set and reducing processing can be incorporated into the inference network model. Second, this framework supports a hybrid processing model where both term- and document-at-a-time processing can be integrated. Third, the safe techniques described in Section 4.2 are still relevant and can be incorporated into the framework. Fourth, complete final scores are calculated for every document in the candidate set, such that the relative rankings are guaranteed correct. Fifth, we can establish an upper bound on the relevance (a lower bound on the final rank) of any document not in the initial candidate set.

Of course, the overall framework is unsafe because we are limiting the set of documents that will be

considered for relevance to the query. The question is, will the population strategies establish an appropriate initial candidate document set.

4.4 Implementation

The techniques discussed above are intended to maximize ROIO by reducing the amount of I/O that must be performed. By definition, the amount of I/O required to access a short inverted list is already optimal. The only way to further reduce I/O for a short inverted list would be to avoid reading the list at all. However, we have seen that short inverted lists correspond to highly discriminating terms that will contribute significantly to the final document ranking. Therefore, I will assume that short inverted lists will always be read during query processing. Furthermore, since a single read will obtain an entire short inverted list, it is not profitable to support disk access of these lists in granularities smaller than the entire list. As such, of the functionality requirements identified earlier during the discussion of optimization techniques, the only one applicable to short inverted lists is keeping the document entries within an inverted list sorted by document identifier.

For long inverted lists, the functionality requirements additionally include the following: separation and isolated access of proximity and belief information, the ability to skip portions of an inverted list when reading it from disk, and maintenance of a list of the documents associated with the top n belief values. Clearly, the implementation should provide different storage structures for short and long inverted lists.

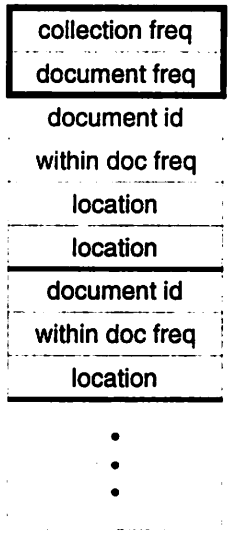

Figure 4: Layout of a short inverted list.

Short lists will be stored in fixed length objects, ranging in size from 16 to 4096 bytes by powers of 2 (i.e., 16, 32, 64, ..., 4096). The 16 byte objects will be allocated in 4096 byte physical segments, while the remaining objects will be allocated in 8192 byte physical segments. A given physical segment will store objects of just one size. The inverted lists in these objects will be laid out as shown in Figure 4, with the collection and document frequencies of the corresponding term at the beginning of the list, followed by a document entry for each document in which the term appears. A document entry consists of a document identifier, the within document frequency of the term in the document, and the location of each occurrence of the term within the document. The collection and document frequencies consume one byte each in lists less than 256 bytes, and two bytes each in larger lists. The rest of the list is compressed as follows. First, the location information associated with each document entry is run-length 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 (document identifiers, within document frequencies, and encoded locations) 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}.

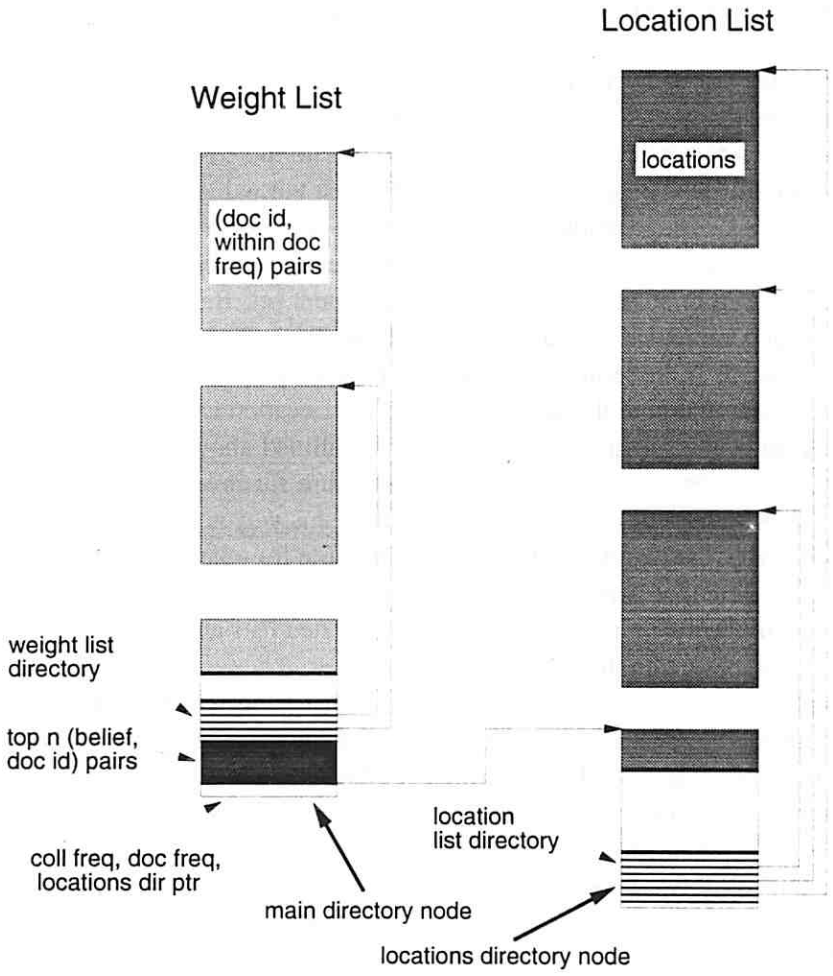

Figure 5: Long inverted list structure.

Long inverted lists (larger than 4096 bytes) will be stored in one or more 8192 byte objects, where each object is allocated in its own segment. If a long inverted list is less than 8192 bytes, it will be stored in a single object and laid out as in Figure 4. Inverted lists larger than 8192 bytes will be stored as shown in Figure 5. Here the inverted list is stored as two distinct lists of information. The Weight List contains document identifier / within document frequency pairs, and the Location List contains the per document occurrence locations. The objects in each list are directly accessed through a directory, one for each list. A directory entry contains a pointer to an object, along with the document identifier 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.

If the within document frequency of a specific document is required, the weight list directory is used to obtain the object in the Weight List that contains this information. The object is then decompressed and

the information is extracted. If the occurrence locations for a specific document are required, the locations directory is used to obtain the object in the Location List that contains this information. To decompress the object and extract the desired information, the document identifiers and within document frequencies for the entries in the Location List object must be obtained from the Weight List. This is accomplished by using the document identifier from the locations directory to index into the Weight List. The entries in both the Weight List and the Location List are stored in the same document identifier order, allowing information for corresponding entries to be extracted simply by synchronizing access to the lists.

The main directory object contains the collection and document frequencies for the term, a pointer to the locations directory, the list of documents associated with the top n belief values in the list, the weight list directory, and, if enough space is available, the tail of the Weight List. The weight list directory and the tail of the Weight List grow towards each other. When there is no more room, the Weight List tail is copied into a new object. When this new object is full, the Weight List tail will again grow into the main directory object, repeating the process. The locations directory object is implemented in a similar fashion.

Access to a long inverted list begins by obtaining the main directory object. At this point, the list of top n beliefs may be accessed to populate the candidate document set. Before any weights are accessed, the weight list directory is decompressed for quick access. If the term is the child of a belief operator, only the Weight List will be accessed. If the term is the child of a proximity operator, the locations directory will be obtained and decompressed, and both the Weight List and the Location List will be accessed.

Besides satisfying all of the functionality requirements outlined above, this storage structure has other attractive features. First, by reserving the complex list structure for inverted lists longer than 8192 bytes, the number of inverted lists that will actually be implemented this way is kept relatively small. Recall from Figure 3 that less than 1% of the inverted lists in the inverted file for a large (2 Gbyte) document collection exceed 8192 bytes. The potential benefits derived from this structure are quite large, however, since the majority of the raw data in the inverted file resides in long inverted lists and the majority of the inverted lists accessed during query processing are long.

Second, this structure will easily support inverted list updates, which occur when new documents are added to, or old documents are deleted from, an existing document collection. When new documents are added, inverted lists must be created for the new terms introduced by the documents, and the inverted lists for existing terms that appear in the new documents must be modified. These modifications involve appending information for the new documents to the existing inverted lists, requiring the ability to grow inverted lists. This process was described by Brown et al. in [6] using an inverted file structure similar to that described here. The main extensions made here are the separation of weight and location information in long inverted lists and the use of directories into the long list objects rather than chaining long list objects in a linked list. Otherwise, document addition is handled the same way, where a small inverted list can grow to fill its current object, relocate to a larger object when the current object is full, and ultimately become a large list, expanding into the more complex large list structure. Appending to the large list structure proceeds as described earlier, such that the majority of the large list is untouched while only the directory and tail objects are modified.

Document deletion is slightly more complicated. Deleting a document involves the deletion of all of the entries for that document in the inverted lists for the terms that appear in that document. For a small inverted list, we simply decompress the existing list, locate the appropriate entry, and recompress into the same object without the entry to be deleted. The newly freed space in the list will appear at the end of the object and be available for future allocation. For a long list, the same procedure is applied to the specific objects that contain the entry for the deleted document, with the addition of possible updates to the directories. Unfortunately, the newly freed space might appear at the end of an object in the middle of the list, resulting in internal fragmentation. This space will never be reused since additions to the list always occur at the end of the list. If the internal fragmentation becomes severe, action may be taken to compact the list. This could be done in an off-line reorganization, or an on-line algorithm could be applied to dynamically reduce

internal fragmentation.

Given the cost of updating the inverted lists during document deletion, individual document deletions should not be applied directly to the inverted lists. Instead, they should be added to a delete list which acts as a filter during query processing. Before the final document ranking for a query is returned to the user, deleted documents are removed from the answer. The deletions are ultimately applied to the inverted lists in a batch. This process can be run during periods of low system demand, or may be triggered by excessive degradation of query processing speed due to the filtering process.

4.5 Evaluation

I plan to evaluate the benefits of the following techniques: separating belief information from proximity information in long inverted lists, providing more direct access into long inverted lists to support intersection optimizations, and the overall strategy for integrating unsafe techniques into the inference network model. For a baseline, I will use document-at-a-time processing with no optimization. The basic hypothesis to be proven in all cases is that the given optimization technique will reduce the number of disk reads required to evaluate the query, translating into a reduction in overall running time.

My evaluation will begin with an assessment of the benefits of the safe optimizations. This will be accomplished by separately adding each of the safe techniques to the baseline implementation and measuring their individual contributions to execution performance.

To evaluate the unsafe technique, the new processing strategy will be incorporated into the baseline system with both of the safe techniques. The unsafe technique requires the specification of a function that determines how the top n belief values in a long inverted list are chosen for population of the candidate document set. It is expected that as n grows, retrieval effectiveness will improve and execution performance will deteriorate. n is the parameter that controls the tradeoff between retrieval effectiveness and execution performance. To measure this tradeoff, I will determine how n should be chosen to obtain a variety of different retrieval effectiveness levels. I will then measure the execution performance of the long inverted list implementation for each of these values of n. The results should suggest a function for determining n that provides improved execution performance with acceptable levels of retrieval effectiveness.

Execution performance will be evaluated by measuring wall clock time and real disk I/O operations. Retrieval effectiveness will be measured using the standard 11pt recall/precision average, with the baseline implementation providing the benchmark retrieval effectiveness against which the unsafe technique will be compared. The document collection used in these experiments will be the largest TREC collection available with queries and associated relevance scores (at least 2 Gbytes).

Another implementation detail with implications for performance is the choice of physical segment size. So far I have assumed that disk block size is a reasonable guideline for physical segment size, since that much data is read with each disk read. However, this may result in unnecessary work to process the physical segment. Consider an object in a long inverted list. Even if we desire information for just a single document in the object, the entire object must be decompressed. A smaller physical segment size, while costing the same in terms of I/O, might provide significant savings in processing time. The issue is essentially the granularity of access into a long inverted list, and is related to the discussion of compression in Section 4.2. Therefore, it would be worthwhile to measure execution performance with 4096 byte physical segments for the long list objects.

5 Related Work

The first body of work related to the research being proposed here is the general technique of supporting information retrieval with a standard database management system. Some of the earliest work was done by

Crawford and MacLeod [13, 33, 12, 34], who describe how to use a relational database management system (RDBMS) to store document data and construct information retrieval queries. Similar work was presented more recently by Blair [4] and Grossman and Driscoll [23]. Others have chosen to extend the relational model to allow better support for IR. Lynch and Stonebraker [32] show how a relational model extended with abstract data types can be used to better support the queries that are typical of an IR system.

In spite of evidence demonstrating the feasibility of using a standard or extended RDBMS to support information retrieval, the poor execution performance of such systems has led IR system builders to construct production systems from scratch. Additionally, most of the work described above deals only with document titles, author lists, and abstracts. Techniques used to support this relatively constrained data collection may not scale to true full-text retrieval systems. I desire to support full-text retrieval with high performance. My approach, while similar in spirit to the above work, differs in both the data management technology chosen to support IR and the extent to which it is applied for that task. Moreover, the functionality requirements imposed by my inverted list implementation are unlikely to be adequately supported in an RDBMS.

Other work in this area has attempted to integrate information retrieval with database management [17, 47]. The services provided by a database management system (DBMS) and an IR system are distinct but complementary, making an integrated system very attractive. The integrated architecture consists of a DBMS component and a custom IR system component. There is a single user interface to both systems, and a preprocessor is used to delegate user queries to the appropriate subsystem. Additionally, the DBMS is used to support the low level file management requirements of the whole system. This architecture is similar to mine in that a separate data management system is used to support the file management requirements of the IR system. However, I am not concerned with integrated support for more traditional data, and my emphasis is on providing data management functionality to support I/O related query optimization.

Efficient management of full-text database indexes has received a fair amount of attention. Faloutsos [18] gives an early survey of the common indexing techniques. Zobel et al. [64] investigate the efficient implementation of an inverted file index for a full-text database system. Their focus is on compression techniques to limit the size of the inverted file index. They also address updates to the inverted file and investigate the different inverted file index record formats necessary to satisfy certain types of queries. My proposed work can be considered an extension of these more traditional inverted list implementations, which simply do not provide the functionality required by the query processing optimizations I am considering.

A more sophisticated inverted list implementation was proposed by Faloutsos and Jagadish [19]. In their scheme, small lists are stored as inverted lists, while large lists are stored as signature files. They have a similar goal of reducing the processing costs for long inverted lists, but their solution is inappropriate for the inference network model. In [20], Faloutsos and Jagadish examine storage and update costs for a family of long inverted list implementations. While they describe linked list implementations for long inverted lists, they do not consider a scheme which supports the fine granularity of access into a long inverted list required by the optimizations considered here.

Harman and Candela [25] use linked lists for a temporary inverted file created during indexing. However, their linked list nodes are quite small, consisting only of a single document posting. Accessing the inverted file in this format during query processing is much too inefficient, so the nodes in a linked list are ultimately conglomerated into a single inverted list before the file is used for retrieval.

Tomasic et al. [53] propose a new data structure to support incremental indexing, and present a detailed simulation study over a variety of disk allocation schemes. The study is extended with a larger synthetic document collection in [49], and a comparison is made with traditional indexing techniques. Their data structure manages small inverted lists in buckets and dynamically selects large inverted lists to be managed separately. Again, they do not support the granularity of access into the long inverted lists required by the optimizations considered here. It is notable that they expect the scheme with the best incremental update performance to have the worst query processing performance due to fragmentation of the long inverted lists. My inverted list implementation will have similar incremental update characteristics, but is actually intended

to provide superior query processing performance.

Wong and Lee [61] describe an inverted list implementation where the inverted list entries are sorted by within document term frequency and each inverted list is divided into pages. The potential similarity contribution of a given inverted list page can be estimated from the inverse document frequency of the term and the maximum within document frequency on the page. The pages are then processed in order of potential similarity contribution. This organization is consistent with the strategy of maximizing ROIO and was shown to produce a high level of retrieval effectiveness with substantially less I/O. While a finer granularity of access is provided into the inverted lists, the organization is incompatible with the inference network model. Access into the inverted lists must be document based in order to ensure that complete scores are calculated for documents.

A similar inverted list structure is described by Persin in [43], where inverted list entries are also sorted by within document term frequency. The emphasis in this work, however, is the efficient storage of the information contained in the lists. A compression strategy is described that allows the new inverted file structure to be stored in less space than that required by an inverted file using document identifier ordering and conventional compression techniques.

Moffat and Zobel [39] describe an inverted list implementation that supports jumping forward in the list using *skip pointers*. This is useful for document based access into the list during conjunctive style processing. However, the purpose of their skip pointers is to provide synchronization points for decompression, allowing just the desired portions of the inverted list to be decompressed. They still must read the entire list from disk. My inverted list structure is more general, allowing fine grained access of a long inverted list at the disk level.

Generic support for storage of large objects has been pursued in the database community. The EXODUS storage manager [10] supports large objects by storing them in one or more fixed size pages indexed by a B+tree on byte address. For example, to access the 12 bytes starting at byte offset 10324 from the beginning of a large object, the object's B+tree would be used to look up 10324 and locate the data page(s) containing the desired bytes. The resultant physical layout of an object in its data pages will be similar to that achieved by my long list structure. The difference is that my directory structure provides customized access to individual pages based on the contents of the page (i.e., document identifier), rather than a byte offset.

The Starburst long field manager [29] supports large objects using a sequence of variable length segments indexed by a descriptor. As an object grows, a newly allocated segment will be twice as large as the previously allocated segment. This growth pattern continues up to some maximum segment size, after which only maximum size segments are allocated. The last segment in the object is trimmed to a page boundary to limit wasted space. This known pattern of growth allows a segment's size to be implicitly determined, eliminating the need to store sizes in the descriptor. A key component of this scheme is the use of a buddy system to manage extents of disk pages from which segments are allocated. This scheme is intended to provide efficient sequential access to large objects, assuming they are typically read or written in their entirety. Unfortunately, this focus is inappropriate for the optimization techniques proposed here where fine grained access to arbitrary locations in a long list is required.

Biliris [2] describes an object store which supports large objects using a combination of techniques from EXODUS and Starburst. A B+tree is used to index variable length segments allocated from disk pages managed by a buddy system. This scheme provides the update characteristics of EXODUS with the sequential access characteristics of Starburst. A comparative performance evaluation of the three schemes can be found in [3]. Again, this scheme does not satisfy my requirements due to the lack of fine grained access based on content into a large object.

The unsafe query optimization techniques have their roots in the upper bound optimizations used to solve the nearest neighbor problem in information retrieval. In this model, a query and the documents in the collection are represented as vectors in an n-dimensional space, where n is the number of terms in the vocabulary. The problem is to find the document closest to the query in this vector space. Distance in the

vector space is defined by the similarity measure used between a document and the query. This is typically some form of dot product between the vectors. The dot product is limited to the terms that appear in the query, so only documents that contain at least one of the query terms need be considered in the nearest neighbor search. Inverted lists are used to identify documents that are the potential nearest neighbor to the query. When a previously unseen document is encountered in an inverted list, the document's representation vector is retrieved to calculate its exact similarity to the query. If this document is closer to the query than the current nearest neighbor, it becomes the new nearest neighbor. When the inverted lists for all of the terms in the query have been processed, the current nearest neighbor is returned as the answer to the query.

Smeaton and van Rijsbergen [51] 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 nearest neighbor, processing may stop. By processing terms in order of increasing inverted list length, they achieve a 40% reduction in the number of similarity calculations required to find the nearest neighbor.

An alternative technique for locating the nearest neighbor uses counters to gradually accumulate a document's similarity to the query. The accumulated similarity is based solely on the information stored in the inverted lists, thus eliminating the need to retrieve the document representation vectors. After all inverted lists have been processed, the nearest neighbor is identified by selecting the maximum similarity from the counters. Perry and Willett [41] show how the upper bound technique can be applied to this processing strategy to reduce main memory requirements. The upper bound on the similarity of a previously unseen document is calculated in the same way as before. If this upper bound is less than the current best similarity for any previously seen document, the new document is not allocated a counter since it cannot be the nearest neighbor. The overall number of counters is reduced, resulting in main memory savings.

This processing strategy can be extended to support full ranking by computing the complete similarity for every document encountered and sorting the set of counters to produce the final ranking. This is the general model assumed for the discussion of unsafe techniques in Section 4.3. In that section, I have already cited related work for each of the unsafe techniques discussed. While some of the previously proposed unsafe techniques may be applied to particular operators in an inference network query tree, it is not clear that applying them in a localized fashion is appropriate. Without a strategy for applying these techniques in a consistent fashion throughout the query tree, the techniques are useless. I have proposed such a strategy, allowing the essence of the unsafe techniques to be incorporated into the inference network model.

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 [48, 27]. 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 my candidate document set, although it is unlikely that reasonable upper bounds for document beliefs could be calculated from the signature information. This is due to the more complex combinations of beliefs supported by the inference network model, as opposed to the simpler dot product similarity measure used by SPIDER.

Finally, properly modeling the size distribution of inverted file index records and the frequency of use of terms in queries is addressed by Wolfram in [59, 60]. He suggests that the informetric characteristics of document databases should be taken into consideration when designing the files used by an IR system. Clearly, this is an underlying theme of the work proposed here, where term frequency and access characteristics are carefully considered throughout.

6 Conclusions

The performance of an information retrieval system is critical to the success of the system. Traditionally, performance has been measured in terms of retrieval effectiveness, or the ability of the system to identify the documents that meet the information need. Another aspect of performance is the speed with which the system can answer queries. This second aspect has often been overlooked in the research community due to an historical lack of large, realistic document collections to support the pursuit of interesting and relevant results. More recently, large document collections have become available to researchers that not only allow the investigation of interesting execution efficiency issues, but actually force them to be addressed.

I have proposed to investigate the execution efficiency aspect of performance for the inference network retrieval model. This model has already demonstrated its prowess in terms of retrieval effectiveness. Unfortunately, the richness of the query language supported by the model presents barriers to the integration of some common optimization techniques that might otherwise by used to improve its execution performance. Using a strategy of maximizing return on I/O, I have identified the two basic functions that must be performed to improve the execution efficiency of this model. First, we must be able to identify the indexing information that will be most useful for evaluating the query. Second, we must be able to access from disk just that information. To accomplish the first task, I have proposed a strategy that allows the essence of previously proposed optimization techniques to be incorporated in a consistent fashion into the inference network model. For the second task, I have described an inverted file implementation based on a persistent object store that provides the functionality required by the proposed optimizations.

Preliminary results have demonstrated the feasibility of an architecture that uses a persistent object store to implement the inverted file in an inference network based document retrieval system. Results from others suggest that certain query optimizations can provide significant execution savings in simpler retrieval models. The research proposed here will evaluate a technique for integrating these optimizations into the inference network model and hopefully result in a system that provides superior execution performance.

References

[1] T. C. Bell, A. Moffat, C. G. Nevill-Manning, I. H. Witten, and J. Zobel. Data compression in full-text retrieval systems. *J. Amer. Soc. Inf. Sci.*, 44(9):508–531, Oct. 1993.

[2] A. Biliris. An efficient database storage structure for large dynamic objects. In *Proc. 8th IEEE Inter. Conf. on Data Engineering*, pages 301–308, Tempe, AZ, Feb. 1992.

[3] A. Biliris. The performance of three database storage structures for managing large objects. In *Proc. of the ACM SIGMOD Inter. Conf. on Management of Data*, pages 276–285, San Diego, CA, June 1992.

[4] D. C. Blair. An extended relational document retrieval model. *Inf. Process. & Mgmnt.*, 24(3):349–371, 1988.

[5] A. Bookstein, S. T. Klein, and D. A. Ziff. A systematic approach to compressing a full-text retrieval system. *Inf. Process. & Mgmnt.*, 28(6):795–806, 1992.

[6] 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 VLDB*, pages 192–202, Santiago, Sept. 1994.

[7] 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*, pages 365–378, Cambridge, UK, Mar. 1994.

[8] C. Buckley and A. F. Lewit. Optimization of inverted vector searches. In *Proc. of the 8th Inter. ACM SIGIR Conf. on Res. and Develop. in Infor. Retr.*, pages 97–110, June 1985.

[9] 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 Sys. Apps.*, Sept. 1992.

[10] M. J. Carey, D. J. DeWitt, J. E. Richardson, and E. J. Shekita. Object and file management in the EXODUS extensible database system. In *Proc. of the 12th Inter. Conf. on VLDB*, pages 91–100, Kyoto, Aug. 1986.

[11] C. W. Cleverdon, J. Mills, and E. M. Keen. Factors determining the performance of indexing systems, vol. 1: Design, vol. 2: Test results. Aslib Cranfield Research Project, Cranfield, England, 1966.

[12] R. G. Crawford. The relational model in information retrieval. *J. Amer. Soc. Inf. Sci.*, 32(1):51–64, 1981.

[13] R. G. Crawford and I. A. MacLeod. A relational approach to modular information retrieval systems design. In *Proc. of the 41st Conf. of the Amer. Soc. for Inf. Sci.*, 1978.

[14] W. B. Croft. Document representation in probabilistic models of information retrieval. *J. Amer. Soc. Inf. Sci.*, 32(6):451–457, Nov. 1981.

[15] W. B. Croft. Experiments with representation in a document retrieval system. *Inf. Tech.: Res. Dev.*, 2(1):1–21, 1983.

[16] W. B. Croft and D. J. Harper. Using probabilistic models of document retrieval without relevance information. *J. Documentation*, 35(4):285–295, Dec. 1979.

[17] J. S. Deogun and V. V. Raghavan. Integration of information retrieval and database management systems. *Inf. Process. & Mgmnt.*, 24(3):303–313, 1988.

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

[19] C. Faloutsos and H. V. Jagadish. Hybrid index organizations for text databases. In *Proc. of the 3rd Inter. Conf. on Extending Database Technology*, pages 310–327, 1992.

[20] C. Faloutsos and H. V. Jagadish. On b-tree indices for skewed distributions. In *Proc. of the 18th Inter. Conf. on VLDB*, pages 363–374, Vancouver, 1992.

[21] E. A. Fox. Characterization of two new experimental collections in computer and information science containing textual and bibliographic concepts. Technical Report 83–561, Cornell University, Ithaca, NY, Sept. 1983.

[22] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, San Mateo, CA, 1993.

[23] D. A. Grossman and J. R. Driscoll. Structuring text within a relational system. In *Proc. of the 3rd Inter. Conf. on Database and Expert Sys. Apps.*, pages 72–77, Sept. 1992.

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

[25] 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.

[26] 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.

[27] D. Knaus and P. Schäuble. Effective and efficient retrieval from large and dynamic document collections. In Harman [24], pages 163–170.

[28] C. Lamb, G. Landis, J. Orenstein, and D. Weinreb. The ObjectStore database system. *Commun. ACM*, 34(10):50–63, Oct. 1991.

[29] T. J. Lehman and B. G. Lindsay. The starburst long field manager. In *Proc. of the 15th Inter. Conf. on VLDB*, pages 375–383, Amsterdam, Aug. 1989.

[30] G. Linoff and C. Stanfill. Compression of indexes with full positional information in very large text databases. In *Proc. of the 16th Inter. ACM SIGIR Conf. on Res. and Develop. in Infor. Retr.*, pages 88–95, Pittsburgh, PA, June 1993.

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

[32] C. A. Lynch and M. Stonebraker. Extended user-defined indexing with application to textual databases. In *Proc. of the 14th Inter. Conf. on VLDB*, pages 306–317, 1988.

[33] I. A. MacLeod. SEQUEL as a language for document retrieval. *J. Amer. Soc. Inf. Sci.*, 30(5):243–249, 1979.

[34] I. A. MacLeod and R. G. Crawford. Document retrieval as a database application. *Inf. Tech.: Res. Dev.*, 2(1):43–60, 1983.

[35] M. E. Maron and J. L. Kuhns. On relevance, probabilistic indexing and information retrieval. *J. ACM*, 7(3):216–244, July 1960.

[36] A. Moffat and J. Zobel. Compression and fast indexing for multi-gigabyte text databases. *Australian Comput. J.*, 26(1):1–9, February 1994.

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

[38] A. Moffat and J. Zobel. Self-indexing inverted files. In *Proc. Australasian Database Conf.*, Christchurch, New Zealand, January 1994.

[39] 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.

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

[41] 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.

[42] M. Persin. Document filtering for fast ranking. In *Proc. of the 17th Inter. ACM SIGIR Conf. on Res. and Develop. in Infor. Retr.*, pages 339–348, Dublin, July 1994.

[43] M. Persin, J. Zobel, and R. Sacks-Davis. Fast document ranking for large scale information retrieval. In *Proc. ADB'94 Inter. Conf. on Applications of Databases*, Vadstena, Sweden, June 1994.

[44] U. Pfeifer, S. Pennekamp, and N. Fuhr. Incremental processing of vague queries in interactive retrieval systems. University of Dortmund internal document, Jan. 1994.

[45] S. E. Robertson and K. Sparck Jones. Relevance weighting of search terms. *J. Amer. Soc. Inf. Sci.*, 27(3):129–146, May 1976.

[46] G. Salton and M. J. McGill. *Introduction to Modern Information Retrieval.* McGraw-Hill, New York, 1983.

[47] L. V. Saxton and V. V. Raghavan. Design of an integrated information retrieval/database management system. *IEEE Trans. Know. Data Eng.*, 2(2):210–219, June 1990.

[48] P. Schäuble. SPIDER: A multiuser information retrieval system for semistructured and dynamic data. In *Proc. of the 16th Inter. ACM SIGIR Conf. on Res. and Develop. in Infor. Retr.*, pages 318–327, Pittsburgh, June 1993.

[49] K. Shoens, A. Tomasic, and H. Garcia-Molina. Synthetic workload performance analysis of incremental updates. In *Proc. of the 17th Inter. ACM SIGIR Conf. on Res. and Develop. in Infor. Retr.*, Dublin, July 1994.

[50] V. Singhal, S. V. Kakkad, and P. R. Wilson. Texas, an efficient, portable persistent store. In *Proc. of the 5th Inter. Workshop on Persistent Object Systems*, pages 11–33, San Miniato, Italy, Sept. 1992.

[51] 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 Res. and Develop. in Infor. Retr.*, pages 83–87, Oakland, CA, 1981.

[52] K. Spark Jones and C. A. Webster. Research in relevance weighting. British Library Research and Development Report 5553, Computer Laboratory, University of Cambridge, 1979.

[53] A. Tomasic, H. Garcia-Molina, and K. Shoens. Incremental updates of inverted lists for text document retrieval. In *Proc. of the ACM SIGMOD Inter. Conf. on Management of Data*, pages 289–300, Minneapolis, MN, May 1994.

[54] H. R. Turtle and W. B. Croft. Inference networks for document retrieval. In *Proc. of the 13th Inter. ACM SIGIR Conf. on Res. and Develop. in Infor. Retr.*, pages 1–24, Sept. 1990.

[55] H. R. Turtle and W. B. Croft. Efficent probabilistic inference for text retrieval. In *Proc. of RIAO'91*, pages 644–661, Barcelona, Apr. 1991.

[56] 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.

[57] H. R. Turtle and J. Flood. Query evaluation: Strategies and optimizations. under review, 1994.

[58] I. H. Witten, A. Moffat, and T. C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images*. Van Nostrand Reinhold, New York, 1994.

[59] D. Wolfram. Applying informetric characteristics of databases to IR system file design, Part I: informetric models. *Inf. Process. & Mgmnt.*, 28(1):121–133, 1992.

[60] D. Wolfram. Applying informetric characteristics of databases to IR system file design, Part II: simulation comparisons. *Inf. Process. & Mgmnt.*, 28(1):135–151, 1992.

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

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

[63] J. Zobel and A. Moffat. Adding compression to a full-text retrieval system. In *Proc. 15th Australian Comp. Sci. Conf.*, pages 1077–1089, Hobart, Australia, Jan. 1992.

[64] J. Zobel, A. Moffat, and R. Sacks-Davis. An efficient indexing technique for full-text database systems. In *Proc. of the 18th Inter. Conf. on VLDB*, pages 352–362, Vancouver, 1992.