Approximating the Top-m Passages in a Parallel Question Answering System

Charles L. A. Clarke
School of Computer Science
University of Waterloo
Waterloo, Canada
claclark@plg.uwaterloo.ca

Egidio L. Terra
School of Computer Science
University of Waterloo
Waterloo, Canada
elterra@plg.uwaterloo.ca

ABSTRACT

We examine the problem of retrieving the top-m ranked items from a large collection, randomly distributed across an n-node system. In order to retrieve the top m overall, we must retrieve the top m from the subcollection stored on each node and merge the results. However, if we are willing to accept a small probability that one or more of the top-m items may be missed, it is possible to reduce computation time by retrieving only the top $k < m$ from each node. In this paper, we demonstrate that this simple observation can be exploited in a realistic application to produce a substantial efficiency improvement without compromising the quality of the retrieved results. To support our claim, we present a statistical model that predicts the impact of the optimization. The paper is structured around a specific application — passage retrieval for question answering — but the primary results are more broadly applicable.

Categories and Subject Descriptors

H.3.4 [**Information Systems**]: Information Storage and Retrieval—*Systems and Software*

General Terms

Design, Experimentation, Measurement, Performance

Keywords

Ranking Queries, Parallel Information Retrieval, Question Answering

1. INTRODUCTION

A *ranking query* retrieves an ordered list of the top-m items from a set, where the items are ranked according to values assigned by a scoring function. For example, information retrieval systems typically return the top-m documents from a collection that are most likely to be be relevant to a natural-language query posed by a user. In some information retrieval applications, items other than documents are retrieved. Question answering systems often use an information retrieval component to return the top-m passages that are most likely to contain the answer to a question [20]. XML information retrieval systems may return document subcomponents as well as complete documents [3, 10, 12]. Ranking queries are also supported in relational databases [1, 8], content-based image retrieval systems [15, 22], and other areas.

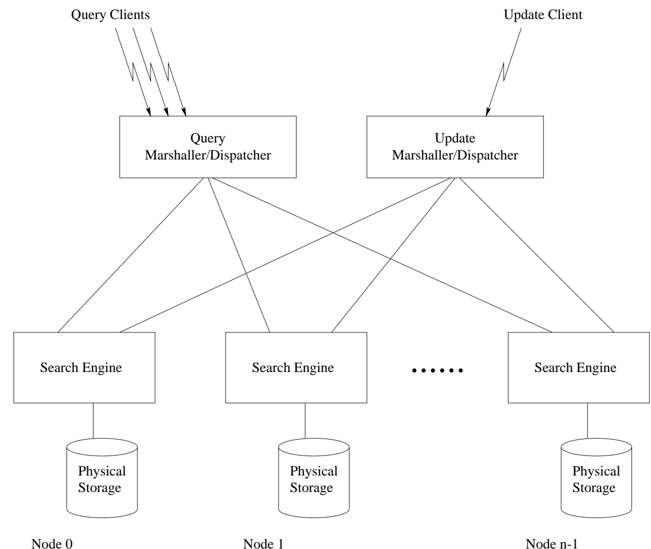

Figure 1: *Architecture of the MultiText information retrieval system.*

In this paper we focus on the efficient execution of ranking queries in parallel information retrieval systems, particularly in the context of question answering. However, nothing precludes the application of the core of this work to ranking queries in other distributed and parallel environments, and we keep the presentation as general as possible. In particular, the work presented in section 4 should be broadly applicable, and the rest of the paper may in part be viewed as an extended example to validate it.

Parallel information retrieval systems are often based on a cluster-of-workstations architecture. The document collection is split into n subcollections that are indexed on different nodes in the cluster [6, 7, 18]. Each node processes

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
CIKM'04, November 8–13, 2004, Washington, DC, USA.
Copyright 2004 ACM 1-58113-874-1/04/0011 ...$5.00.

a query independently, retrieving the top documents from its local subcollection. The results of these independent searches, which are small in size and require little bandwidth to communicate, are gathered centrally and merged to create a final result set.

As an example, Figure 1 shows the architecture of the MultiText information retrieval system, which was used to generate the experimental results reported in this paper. External clients contact the query marshaller/dispatcher, which forwards the client's query to search engines running on each node in the cluster, combines the results, and reports them back to the client. The query marshaller/dispatcher may run on any node in the system, or may run on a dedicated node. The update marshaller/dispatcher performs a similar role for an external update agent, as well as implementing the system's load balancing and fault tolerance strategies [6]. In this paper, we address only search performance; updates are not further discussed.

To guarantee globally that the top-m items from the collection are retrieved, each node must retrieve the top-m items from its local subcollection. The top-m items from each subcollection are required to handle the case that a single node contains the top-m items from the collection as a whole. This case would not be unusual if items were distributed according to their content or source. However, if the top-m items are randomly distributed across the system, the probability that all of them will be located on the same node is extremely small: $(1/n)^m$. In many cases, returning the top $k < m$ items from each node will retrieve the top m globally with high probability. If we are willing to tolerate a small probability of missing one or more of the top m, each node could compute only the top $k < m$, providing an opportunity to reduce overall execution time.

Whether or not we can realize a substantial gain in efficiency by computing the top $k < m$ on each node, rather than the top m, depends on the specific information retrieval algorithm. In the case of many standard information retrieval algorithms it is not obvious that any benefit would result, except in the extreme case when m is large enough that the time to communicate query results across the network plays a significant role in the overall query time. In textbooks and research literature, information retrieval techniques are often expressed as formulae that are applied to each document containing one or more terms from the user's query. If these algorithms are implemented naïvely, the size of the result set (m) has no impact on efficiency, since a score for each document must still be computed. With a few exceptions [23], the impact of result set size on efficiency has not been a concern of information retrieval research.

In this paper, we present an passage retrieval algorithm for question answering with an average execution time that depends on the size of the result set. A statistical model is used to estimate the impact of retrieving $k < m$ passages from each node. For this application, surprisingly small values of k can be used, substantially reducing retrieval times without compromising the quality of the results.

1.1 Related Work

The efficient implementation of ranking queries is a topic of current interest in the database community, and in much of this work the number of items (m) is considered to be a key parameter. In particular, over the past few years, the database community has devoted attention to the problem of optimizing ranked query with joins. In general, these queries are comprised of lists of ranked data and a join predicate specified by the user. Carey et al. [2] extended SQL to incorporate a new operator that allows a user to specify the desired depth for a join operation. In cases where the matching between query and the database is not exact, the results are normally scored by similarity, and in such settings join operations also address ranked data. Natsev et. al. [17] investigate multimedia queries, based on similarity scores, and proposed algorithms to support incremental joins. In traditional relational databases, the limitations of SQL require a join to be fully executed before sorting to compute the final ranking. This is not desirable since the size of the join is exponential in the number of ranked lists in the projection (with base equals to the geometric mean). Recently, rank-aware operators were proposed and they allow queries with join predicates to be optimized without the need to execute a complete join [1, 8].

The work presented in this paper has application to other passage retrieval algorithms [9, 11, 13, 14, 16, 20]. Unlike the MultiText algorithm, which extracts passages directly from the collection, many of these algorithms operate by first retrieving the top documents and then scanning them to identify high-scoring passages. Thus, passage retrieval time grows in proportion to the number of documents. Some systems retrieve and scan as many as the top 1000 documents. In a multi-node system, reducing the number of documents scanned on each node would produce a corresponding decrease in processing time.

Witten et al. [23] provide a through summary of efficient implementation techniques for document-oriented information retrieval. They discuss the problem of identifying the top-m documents during the final stage of query processing, after scores have been assigned to documents, and propose a selection algorithm to reduce execution time when m is small relative to the number of documents.

1.2 Organization of the Paper

After a brief introduction to question answering (QA) systems, we describe an algorithm that computes the top-m passages from a question. This algorithm has been used successfully as a component in the MultiText QA system since 2000 [4, 5]. In terms of its retrieval *effectiveness*—its ability to identify passages containing answers to a question — it is comparable to the best known algorithms [20]. Here, we focus on its *efficiency* rather than its effectiveness, measuring the impact of result set size on execution time and demonstrating the potential for overall performance improvement. Section 4 presents a statistical model of the distribution of items in an n-node system that allows us to determine retrieval probabilities as result set sizes are varied. Section 5 applies this model to the MultiText passage retrieval algorithm, examining its impact on both efficiency and effectiveness.

1.3 Terminology and Assumptions

For clarity, this section summarizes the key terminology used throughout the paper. We assume the context of an information retrieval or database system whose goal is to identify the *top-m items* from a large *collection*, which may consist of documents, passages, XML entities, database records, or other objects, depending on the application. We refer these m items as the *target set* or the *target items*.

We assume the collection is divided into n *subcollections* of roughly equal size, where each subcollection is stored and searched on a separate node of an n-node distributed or parallel computer. We assume the items are distributed randomly (uniformly and independently) across the n nodes.

We assume the system processes a ranking query by retrieving the top $k \leq m$ items from each subcollection, merging the results, and returning the top m. We refer to k as the *retrieval depth.* Items are ranked according to a real-valued *scoring function* that can be applied to each item independently, so that an item will be assigned the same score regardless of the other items in the subcollection that contains it.

2. QUESTION ANSWERING

We use the "factoid" question answering task as a vehicle to demonstrate the validity and potential benefits of our work. The goal of this task is to produce an short, factual answer in response to a question posed by a user. For example, the user might pose the question, "What mythical Scottish town appears for one day every 100 years?" and receive the answer "Brigadoon", or pose the question "Who was Galileo?" and receive the answer "astronomer and pioneer of modern physical science." Most trivia questions provide an example of this task.

Over the past several years, a considerable body of work has been published concerning this task [4,5,9,13,14,16,19–21]. This work has been encouraged by the inclusion of a question answering task in the TREC evaluations conducted annually by the National Institute of Standards and Technology. The evaluation reported in this paper is based on a set of 1,732 questions taken from the TREC QA experiments conducted between 2000 and 2003.

Many QA system, including the MultiText system, operate according to the following general plan: 1) The question is analyzed to identify key terms, the type of answer expected, and related information. 2) A query is formulated from the question and submitted to an information retrieval system that returns passages that may contain the answer to the question. 3) The passages are processed to extract possible answers.

While QA systems differ greatly in the details of how this plan is implemented, and most incorporate other steps and strategies, these three steps are usually present. Descriptions of many QA systems can be found in the current and past proceedings of TREC[1]. Strzalkowski and Harabagiu [19] provide an overview of current QA research. Tellex et al. [20] survey passage retrieval algorithms for question answering and evaluate their relative effectiveness.

3. PASSAGE RETRIEVAL FOR QA

Complete details of the MultiText QA system, including a discussion of its question analysis and answer extraction components, as well as a description of its passage retrieval algorithm, appear elsewhere [4]. In this section, we reproduce the core of the passage retrieval algorithm. This description is provided to illustrate how the performance of an information retrieval algorithm can vary with the size of the target set. In addition, we extend this description by examining the efficiency of the algorithm as the retrieval depth is varied.

[1] `trec.nist.gov`

Unlike many passage-retrieval algorithms, the MultiText passage retreival algorithm does not return predefined passages, such as paragraphs, sentences or fixed-size windows of text. Instead, it can retrieve any substring of any document in the subcollection stored on a given node. Since passages, rather than documents, are the products of the retrieval algorithm, the primary view of each subcollection is as a single long string, consisting of all the documents in the subcollection concatenated together. Under this view, a subcollection $\mathcal{C}$ can be treated as an ordered sequence of terms:

$$\mathcal{C} = c_1\ c_2\ c_3 ... c_N.$$

A passage from $\mathcal{C}$ is represented by an *extent*, an ordered pair of coordinates (u, v) with $1 \leq u \leq v \leq N$, that corresponds to the subsequence of $\mathcal{C}$ beginning at position u and ending at position v

$$c_u\ c_{u+1}\ c_{u+2} ... c_v.$$

A query Q is generated from the original question and takes the form of a term set:

$$Q = \{t_1,\ t_2,\ t_3, ...\}.$$

An extent (u, v) *satisfies* a term set $T \subseteq Q$ if the subsequence of $\mathcal{C}$ defined by the extent contains at least one term matching each of the terms from T. The determination that a query term t_i matches a collection term c_j may take into account stemming, synonym expansion and similar transformations. In the actual MultiText implementation a term may also be a phrase or a structured query, but for simplicity we ignore these extensions in the description that follows.

An extent (u, v) is a *cover* for T if (u, v) satisfies T and the subsequence corresponding to (u, v) contains no subsequence that also satisfies T. That is, there does not exist an extent $(u', v') \neq (u, v)$ with $u \leq u' \leq v' \leq v$ that also satisfies T.

As an example, for TREC question 1462 ("Where is the oldest synagogue in the United States?") the question analysis step produces the three-term query:

```
"oldest" "synagogue"
"u.s"+"usa"+"american"+"united.states"
```

The last term is a disjunction of words and phrases that expand on the question phrase "United States". A typical fragment retrieved by this query is given in Figure 2. The term cover is shown in **boldface**. Possible answers, based on question type, are shown in *italics*.

While the focus is on passage retrieval, the locations of document boundaries are retained for post-retrieval filtering of passages. First, we exclude extents that cross document boundaries. The co-occurrence of question terms in such an extent may be purely accidental, and a lower scoring passage found at the end of one document or the beginning of the other may be a better choice. Second, we retain only the highest scoring extent from each document. The answer extraction step of the MultiText system depends on the independent appearance of candidates in multiple passages, and the terms appearing in a second passage from the same document are unlikely to exhibit this independence.

3.1 Passage Scoring Function

Given an extent (u, v) that is a cover for a term set $T = \{t_1,\ t_2, ...\}$ we wish to compute a score for the extent with respect to T that reflects the likelihood that an answer to

...The \$350,000 makeover is expected to be completed by December. The structure is the **oldest synagogue under the American** flag to have never missed Sabbath services. It is the third-oldest synagogue in the *Western Hemisphere* after a temple built in 1732 on the *Dutch Caribbean island of Curacao* and a 1763 synagogue in *Newport, R.I.* The congregation also boasts that its synagogue held the first confirmation ceremony for Jewish youth in the *Western Hemisphere*. The historical significance...

Figure 2: *A typical fragment retrieved for TREC question 1462* — the text in boldface was identified by the passage retrieval algorithm; possible answers are shown in *italics*.

the question is contained in the extent or appears in its close proximity. Our score for an extent of length $l = v - u + 1$ containing the terms $T \subseteq Q$ is:

$$\sum_{t \in T} \log(N/f_t) \; - \; |T| \log(l) \qquad (1)$$

where f_t is the total number of times t appears in the subcollection and N is the sum of the lengths of all documents in the subcollection.

Equation 1 assigns higher scores to passages whose probability of occurrence is lower. While a higher score does not directly imply a greater likelihood that the answer will appear in close proximity, but empirical evidence suggests that this relationship holds.

3.2 Passage Retrieval Algorithm

Given a query Q we generate $\mathcal{J}$, the set of all covers for all subsets of Q, and rank them using equation 1. We discard all but the highest-ranking cover from each document. Of those remaining, the top m are processed to extract possible answers.

Implementation of this passage retrieval and ranking technique depends on a fast algorithm to compute all covers of all subsets of Q. An extent (u, v) is said to *i-satisfy* a query Q if the subsequence of $\mathcal{C}$ defined by the extent contains exactly i distinct terms from Q. An extent (p, q) is an *i-cover* for Q if and only if it i-satisfies Q and does not contain a shorter extent that also i-satisfies Q. Below we present an algorithm to generate $\mathcal{J}_i$, the set of all i-covers of Q, in time $O(|Q| \cdot |\mathcal{J}_i| \cdot \log(N))$. The set of covers for all subsets of Q is simply the union of the i-covers

$$\mathcal{J} \; = \; \bigcup_{i=1}^{N} \mathcal{J}_i.$$

There are $O(N^2)$ different extents that might legitimately be returned by a passage-retrieval algorithm, and the significance of the extents in J may not be immediately obvious. If all $O(N^2)$ extents were ranked using equation 1, some of the extents not in $\mathcal{J}$ may be ranked before some of the extents in $\mathcal{J}$. However, from the definition of i-cover and from equation 1, any extent not in $\mathcal{J}$ has at least one higher scoring passage from $\mathcal{J}$ nested within it. Such an extent need not be generated. It would be eliminated from the final ranked list of passages, since it either occurs in a document with a higher ranking passage (the nested passage) or overlaps a document boundary.

The efficient generation of i-covers depends on specific support from the underlying index structures that record locations of terms within the subcollection. This support takes the form of two *access functions* $r(t, w)$ and $l(t, w)$ that return positions in the subcollection term sequence $c_1, ..., c_N$.

```
    Cover(Q, i, w) ≡
1       Let t_1, ..., t_|Q| be the elements of Q.
2       for j ← 1 to |Q| do
3           R[j] ← r(t_j, w)
4       end for
5       v ← ith largest element of R
6       Q' ← ∅
7       for j ← 1 to |Q| do
8           if R[j] ≤ v then
9               Q' ← Q' ∪ {t_j}
10          end if
11      end for
12      Let t'_1, ..., t'_|Q'| be the elements of Q'.
13      for j ← 1 to |Q'| do
14          L[j] ← l(t'_j, v)
15      end for
16      u ← smallest element of L
17      return (u, v)
```

Figure 3: *Passage Retrieval* — Given a query Q, a ranking level i, and a corpus position w, the cover generation procedure generates the first i-cover for Q starting at or after w. The procedure calls the l and r access functions to generate term positions.

Both access functions take a term t and a position in the term sequence w as arguments and return results as follows:

$$r(t,w) = \begin{cases} v & \text{if } \exists\, c_v \text{ matching } t \text{ such that } w \le v \\ & \quad \text{and } \nexists\, c_{v'} \text{ matching } t \\ & \quad\quad \text{such that } w \le v' < v \\ N+1 & \text{otherwise;} \end{cases}$$

and

$$l(t,w) = \begin{cases} u & \text{if } \exists\, c_u \text{ matching } t \text{ such that } w \ge u \\ & \quad \text{and } \nexists\, c_{u'} \text{ matching } t \\ & \quad\quad \text{such that } w \ge u' > u \\ 0 & \text{otherwise.} \end{cases}$$

Informally, the access function $r(t,w)$ returns the position of the first occurrence of the term t located at or after position w in the term sequence. If there is no occurrence of t at or after position w, then $r(t,w)$ returns $N+1$, one position beyond the end of the subcollection. Similarly, the access function $l(t,w)$ returns the position of the last occurrence of the term t located at or before position w in the term sequence. If there is no occurrence of t at or before position w, then $l(t,w)$ returns 0.

The $r(t,w)$ and $l(t,w)$ access functions may be easily implemented using file structures that can resolve a call to either access function in $O(\log N)$ time. The MultiText QA System uses an extended inverted-index file structure originally developed to support retrieval from structured text. These file structures store a sorted list of positions for each term, organized to allow the list to be efficiently addressed by position w, directly supporting the access functions and permitting portions of the list to be skipped when possible.

The generation of i-covers is achieved through calls to the r and l access functions. The procedure *Cover* (Figure 3) takes as its arguments a set of query terms Q, a ranking level i, and a term sequence position w, and generates the first i-cover for Q that starts at or after w.

The loop over lines 2–4 of Figure 3 calls the access function $r(t_j, w)$ for each term $t_j \in Q$, recording the results in $\mathcal{R}$, an array of positions within the subcollection. For each term $t_j \in Q$, $1 \le j \le |Q|$, the position of its first occurrence at or after w is assigned to $\mathcal{R}[j]$. At line 5, the variable v is assigned the ith largest element of $\mathcal{R}$. From the definition of $r(t,w)$, the extent (w,v) i-satisfies Q, and any extent (w,v') with $w \le v' < v$ will not i-satisfy Q. Therefore, the first i-cover for Q starting at or after position w ends at v.

Lines 6–11 construct the set Q' consisting of the i terms from Q that appear in the interval of the subcollection associated with (w,v). The loop over lines 13–15 calls the access function $l({t'}_j, v)$ for each term ${t'}_j \in Q'$. For each term ${t'}_j \in Q'$, $1 \le j \le i$, the position of its last occurrence at or before v is assigned to the array element $\mathcal{L}[j]$. At line 16, the integer variable u is assigned the smallest element of $\mathcal{L}$. From the definition of $l(t,w)$, the extent (u,v) i-satisfies Q, and any extent (u',v) with $u < u' \le v$ will not i-satisfy Q. Therefore, (u,v) is an i-cover for Q.

At line 17, (u,v) is returned to the caller. If Q has no i-cover starting after the specified w, the extend $(N+1, N+1)$ is returned. Generating a single i-cover requires $|Q|+i = O(|Q|)$ calls to access functions.

By definition, i-covers do not nest. It follows immediately that no two i-covers can share a common starting position. The set $\mathcal{J}_i$ of all i-covers of Q can therefore be generated by successive calls to the *Cover* procedure:

$\mathcal{J}_i \leftarrow \emptyset$
$(u,v) \leftarrow Cover(Q,i,0)$
while $u \le N$ **do**
  $\mathcal{J}_i \leftarrow \mathcal{J}_i \cup \{(u,v)\}$
  $(u,v) \leftarrow Cover(Q,i,u+1)$
end while

The generation of $\mathcal{J}_i$ requires $O(|Q| \cdot |\mathcal{J}_i| \cdot \log(N))$ time. Overall, generating $\mathcal{J}$, the set of all covers for all subsets of Q requires $O(|Q| \cdot |\mathcal{J}| \cdot \log(N))$ time.

The top scoring elements of $\mathcal{J}$ from distinct documents become the raw material for the answer extraction step. Since an element of $\mathcal{J}$ may cover only query terms, it is expanded by including text at the beginning and end to provide context. Up to n words of text may be added to both the beginning and end. In most cases we use $n = 100$. After expansion, the result is a set of passages. Passages from each node are merged and the result becomes the input to the answer-extraction step. Within each passage, the location of the original cover is also marked as a "hotspot" for use during answer extraction.

3.3 Efficiency Heuristics

In practice, all elements of $\mathcal{J}$ may not be needed, since only the top k elements of $\mathcal{J}$ from distinct documents must be generated on each node. To this point, the efficiency of the algorithm does not depend on the retrieval depth. However, since the algorithm proceeds in stages, computing $\mathcal{J}_i$ before $\mathcal{J}_{i-1}$, we can apply two heuristics to reduce the work done in later stages and halt the process when we can guarantee that the top k have been found.

The algorithm begins the ranking process by generating $\mathcal{J}_{|Q|}$, the set of covers for all terms of Q, and then generates $\mathcal{J}_{|Q|-1}$ down to $\mathcal{J}_1$ in succession. Before the generation of $\mathcal{J}_i$ for a particular i, it may be the case that k or more covers have already been generated, with the lowest scoring cover having score x. Using equation 1 with the length of the covering extent set to zero ($l = 0$), Q can be checked to determine the largest score that can be obtained from i of its terms. If this score does not exceed x, the cover generation process may be halted.

For term t, define $s(t) = \log(N/f_t)$. Assume that the terms of the query $Q = \{t_1,\ t_2,\ t_3, ...\}$ are labeled such that $a < b$ implies $s(t_a) > s(t_b)$. At the start of stage i, before the computation of $\mathcal{J}_i$ begins, we check

$$\sum_{a=1}^{i} s(t_a) - i < x$$

If so, any cover generated by this or later stages will not form part of the final result set, and the algorithm may be halted.

Similarly, it may be the case that a particular $t \in Q$ cannot be combined with any other $i-1$ terms from Q to produce a cover with score greater than x. In this case, t can be eliminated from further consideration. For example, during stage i this property may be true for term t_j, $j > i$, so that

$$\sum_{a=1}^{i-1} s(t_a) + s(t_j) - i < x$$

In which case t_j may be removed from the term set before the computation of $\mathcal{J}_i$ is begun.

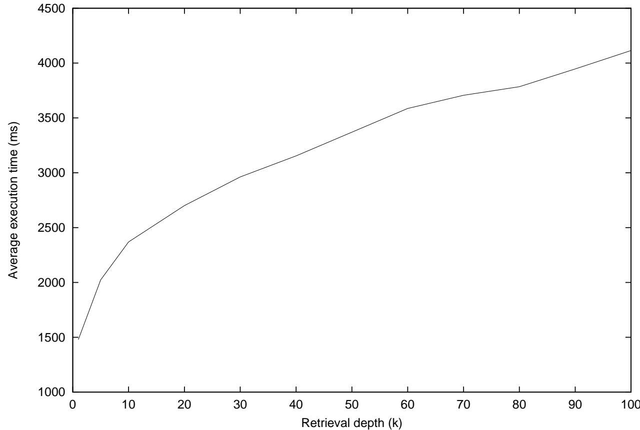

Figure 4: *Efficiency* — average query execution time.

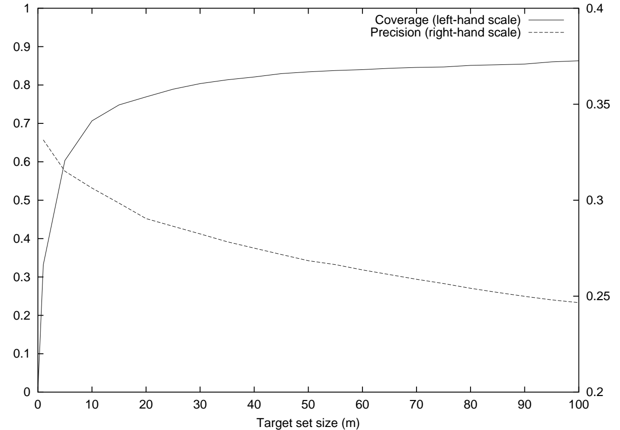

Figure 5: *Effectiveness* — coverage and precision.

3.4 Measurements

To evaluate the algorithm, including the benefits of the heuristics outlined in the previous section, we used a set of 1,732 questions from the 2000-2003 TREC question answering experiments. Associated with each question is one or more answer patterns, specified as regular expressions. We assume that a passage answers a question if it contains a match to one or more of its associated patterns. This question set includes all TREC questions for which answer patterns were available, with the exception of a small number of questions from TREC-9 that were minor re-wordings of other TREC-9 questions.

The questions are first processed by the question analysis component of the MultiText QA system to formulate queries. The queries were then executed over a terabyte of Web data crawled from the general Web during 2001. This data was divided into 36 subcollections of roughly 28GB each, distributed across a cluster-of-workstations.

Figure 4 plots the average query execution time as a function of target set size, with the value of m ranging between 1 and 100. These execution times were measured on a single node in the cluster, a uniprocessor machine with an AMD Athlon running at 1.4Ghz and containing 512 Mbytes of RAM. The disk is a conventional ATA under an IDE interface.

Execution time increases with k, with greater rates of change seen at lower values of k. The execution time for $k = 100$ is twice the time for $k = 5$.

Figure 5 plots two effectiveness measures for these questions. For a given m, *coverage* indicates the ratio of questions that have answers in the top-m passages. As m increases, coverage increases, exceeding a value of 80% at $m = 30$ and continuing to grow slowly after that. *Precision* indicates the ratio of passages in the top-m that contain answers. Precision drops as m increases. Both coverage and precision are important to the performance of the MultiText answer extraction component. Typically, the top $m = 40$ passages are passed to the answer extraction component, providing a reasonable combination of coverage and precision.

4. MODEL

In a multi-node environment, we can exploit the relationship between query execution time and retrieval depth (k) to improve overall performance. Our goal is to return the top-m items across the system as a whole. As stated earlier, the only way to achieve this goal with 100% certainly is to retrieve the top-m items from each node and merge these results. However, if we are willing to relax the requirements slightly, and accept a small probability that one or more of the target items will be missed, we can reduce the number of items that must be retrieved from each node, improving average query time.

In this section, we model the placement of items on nodes in order to quantify the relationship between the probability of failure and the retrieval depth. We assume that items are randomly distributed across the n nodes, uniformly and independently, and that the total number of items is large enough that we can safely ignore the possibility of a node containing less than m items.

The probability that a particular target item will be located on a particular server is $1/n$, and thus the probability that a particular server contains exactly l of the m target items is given by the binomial distribution:

$$b(n, m, l) \; = \; \left(\begin{array}{c} m \\ l \end{array} \right) \left(\frac{1}{n}\right)^{l} \left(1 - \frac{1}{n}\right)^{m-l}$$

Let $p(n, m, k)$ be the probability that all of the m target items will be found by retrieving the top k from each of the n nodes. A recursive formula for $p(n, m, k)$ can be developed by observing that if l items are stored on a particular node ($0 \le l \le k \le m$), then $p(n, m, k)$ depends on the distribution of the remaining $m - l$ target items across the other nodes, $p(n - 1, m - l, k)$. Combining this observation with some simple boundary cases gives:

$$p(n, m, k) \; = \; \begin{cases} 1 & \text{if } m \le k; \\ 0 & \text{if } n = 1 \text{ and } m > k; \\ \displaystyle\sum_{l=0}^{k} b(n, m, l) p(n - 1, m - l, k) & \text{otherwise.} \end{cases}$$

This equation can be solved through the application of dynamic programming.

Given this equation, we can choose a probability threshold (say 95%) and determine a minimum value for k that gives us at least that probability of retrieving the top-m items in an n-node system. Figure 6 plots the minimum k for various values of n and m with a probability threshold of 95%; Figure 7 plots the minimum k with a probability threshold of 99.9%.

On an eight-node system, a retrieval depth of $k = 11$ will return the top $m = 40$ items with greater than 95% probability, and a retrieval depth of $k = 14$ will retrieve the top $m = 40$ items with greater than 99.9% probability. On an 64-node system, a retrieval depth of $k = 7$ will return the top $m = 100$ items with greater than 95% probability and a retrieval depth of $k = 9$ will retrieve the top $m = 100$ items with greater than 99.9% probability.

We can extend our model to consider the problem from a different perspective, by fixing the value of k and determining how large a target document set we expect to retrieve. In other words, for a given k, we may determine a value for m such that we expect to retrieve all of the top-m items.

For a given n, we define the random variable M_k as the size of the target set returned when the top k items from each node are retrieved and merged. Thus, when $M_k = m$ the result set includes all of the top m items, but not the item with rank $m + 1$.

Let $q(n, j, k)$ be the probability of retrieving exactly the top j in an n-node system with retrieval depth k. By "exactly the top j" we mean that all of the items ranked 1 to j are retrieved, but the item ranked $j+1$ is not retrieved. Since $p(n, j, k)$ is the probability that *at least* the top j will be retrieved, we observe that $q(n, j, k) = p(n, j, k) - p(n, j+1, k)$. We also observe that $q(n, j, k) = 0$ when $j > n \cdot k$. Combining these observations with a little algebra allows us to compute the expected value of M_k:

$$\begin{aligned} E[M_k] &= \sum_{j=1}^{\infty} j \cdot q(n, j, k) \\ &= \sum_{j=1}^{n \cdot k} j \cdot (p(n, j, k) - p(n, j + 1, k)) \\ &= \sum_{j=1}^{n \cdot k} p(n, j, k) \end{aligned}$$

Figure 8 plots the minimum k against the expected target set size. On an eight-node system, a retrieval depth of $k = 8$ gives an expected target set size of 40, and a retrieval depth of $k = 18$ gives an expected target set size of 100. On a 64-node system, a retrieval depth of $k = 3$ gives an expected target set size of 40, and a retrieval depth of $k = 5$ gives an expected target set size of 100.

Either a specific probability threshold or the expected target set size may be used as a criterion to select a value for k. This selection does not depend on characteristics of the application, other than those implied by the basic assumptions listed in Section 1.3. For particular applications, smaller values of k may produce acceptable results. In the next section, we apply the model to the passage retrieval application, examining its impact on both efficiency and effectiveness.

5. IMPACT

In the case of passage retrieval for QA, the values for k produced by the criteria in the previous section are fairly

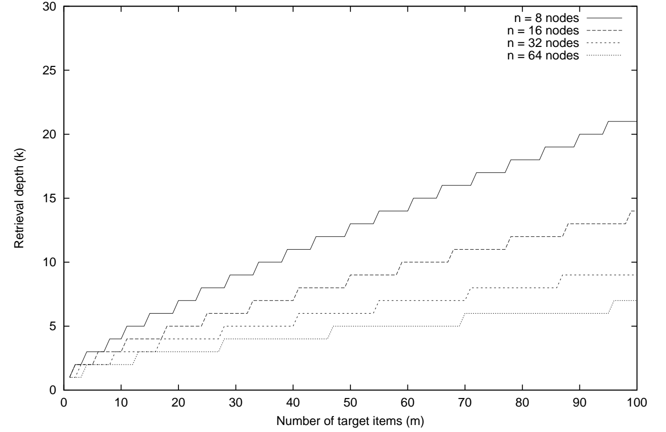

Figure 6: ***Minimum retrieval depth with probability threshold 95.0%.***

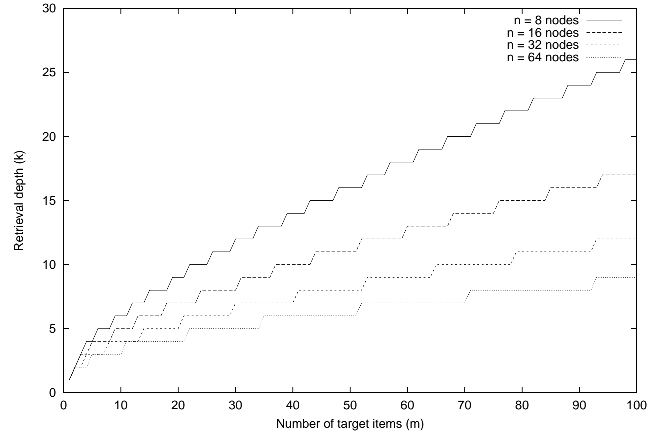

Figure 7: ***Minimum retrieval depth with probability threshold 99.9%.***

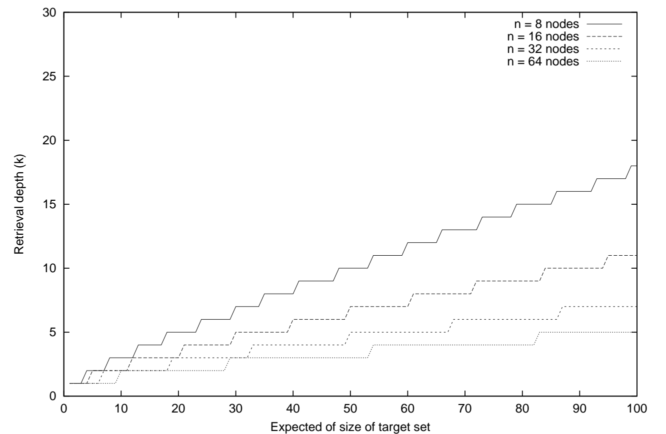

Figure 8: ***Minimum retrieval depth by expected target set size ($E[M_k]$).***

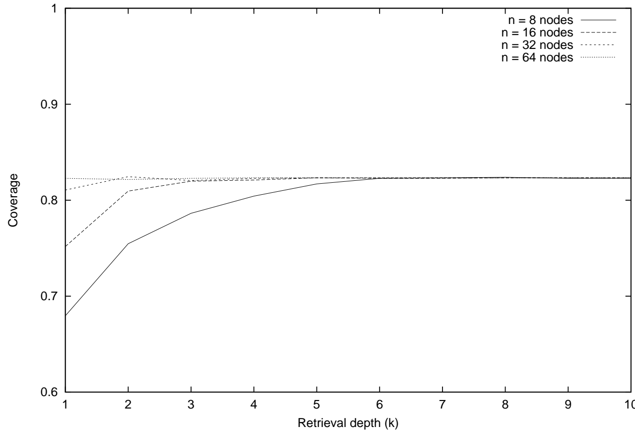

Figure 9: *Coverage* — target set size $m = 40$

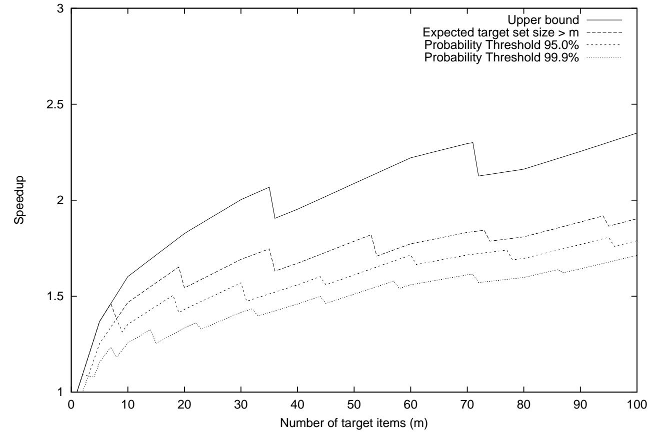

Figure 11: *Speedup*

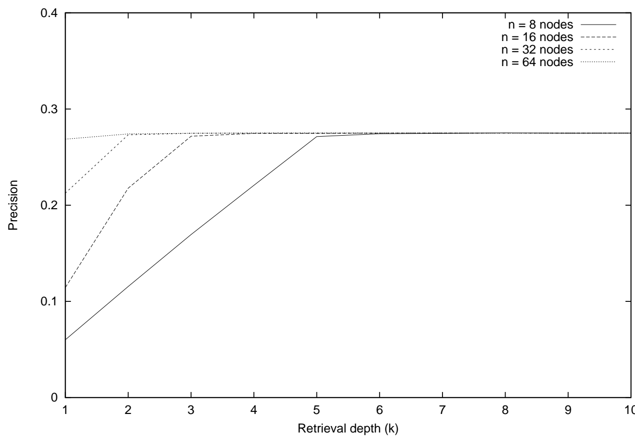

Figure 10: *Precision* — target set size $m = 40$

conservative. Figures 9 and 10 plot coverage and precision as a function of k for four different cluster sizes. In both figures we use a fixed target set size of $m = 40$. For these figures, various cluster sizes were simulated from the top-1000 passages for each question, by randomly distributing passages to nodes. These top-1000 passages were generated on the cluster described in section 3.4, where $n = 36$. As indicated in these figures, if $n \cdot k > m$ the effectiveness of the system is essentially indistinguishable from the baseline given in Figure 5. For this application, $k = \lceil m/n \rceil$ would be an acceptable choice.

Figure 11 shows the system speedups obtained when various criterion are used to select k. These speedups are based on the measured execution times plotted in Figure 4, with a cluster size of $n = 36$. For a given criterion, the value of m is used to select a value for k, and the speedup is computed as the ratio of the baseline execution time to the execution time with a retrieval depth of k. The upper bound given in the figure corresponds to a choice of $k = \lceil m/n \rceil$. The other curves correspond to the selection criteria from Section 4 as illustrated in Figures 6 to 8. The jaggedness in the curves is caused by shifts in the value of k as the value of m is increased. For $m = 40$, speedups range from 1.45 to 1.95.

By default, the MultiText system uses a probability threshold of 95.0% for all ranked queries. The client supplies a value for m to the query marshaller/dispatcher (Figure 1), which converts it to the corresponding retrieval depth and passes this value to the individual search engines along with the query.

6. CONCLUSION

This work is based on a simple observation that applies to the processing of any top-m query in an n-node system, when the appropriate assumptions are satisfied. If we accept a small probability that one or more of the top-m items may be missed, it is possible to reduce computation time by retrieving only the top $k < m$ from each node. The contribution of this paper is a demonstration that this observation may be beneficially applied in a realistic environment, along with the provision of a statistical model to guide its application.

7. REFERENCES

[1] Nicolas Bruno, Surajit Chaudhuri, and Luis Gravano. Top-k selection queries over relational databases: Mapping strategies and performance evaluation. *ACM Transactions on Database Systems*, 27(2):153–187, 2002.

[2] Michael J. Carey and Donald Kossmann. On saying "Enough already!" in SQL. In *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data*, pages 219–230, 1997.

[3] David Carmel, Yoelle S. Maarek, Matan Mandelbrod, Yosi Mass, and Aya Soffer. Searching XML documents via XML fragments. In *26nd Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 151–158, Toronto, Canada, July 2003.

[4] C. L. A. Clarke, G. V. Cormack, T. R. Lynam, and E. L. Terra. Question answering by passage selection. In Tomek Strzalkowski and Sanda Harabagiu, editors, *Advances in Open Domain Question Answering*. Kluwer Academic Press, 2004.

[5] Charles L. A. Clarke, Gordon V. Cormack, and Thomas R. Lynam. Exploiting redundancy in question answering. In *24th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 358–365, New Orleans, September 2001.

[6] Charles L. A. Clarke, Philip L. Tilker, Allen Quoc-Luan Tran, Kevin Harris, and Antonio S. Cheng. A reliable storage management layer for distributed information retrieval systems. In *12th ACM International Conference on Information and Knowledge Management*, pages 207–215, New Orleans, 2003.

[7] David Hawking, Nick Craswell, and Paul Thistlewaite. Overview of the TREC-7 very large collection track. In *7th Text REtrieval Conference*, Gaithersburg, Maryland, November 1998.

[8] Ihab F. Ilyas, Rahul Shah, Walid G. Aref, Jeffrey Scott Vitter, and Ahmed K. Elmagarmid. Rank-aware query optimization. In *Proceedings of the ACM SIGMOD/PODS 2004 Conference*, Paris, 2004.

[9] A. Ittycheriah, M. Franz, W-J Zhu, A. Ratnaparkhi, and R.J. Mammone. IBM's statistical question answering system. In *9th Text REtrieval Conference*, Gaithersburg, Maryland, November 2000.

[10] Jaap Kamps, Maarten de Rijke, and Börkur Sigurbjörnsson. Length normalization in XML retrieval. In *27th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, Sheffield, UK, 2004.

[11] Marcin Kaszkiel and Justin Zobel. Effective ranking with arbitrary passages. *Journal of the American Society of Information Science*, 52(4):344–364, 2001.

[12] G. Kazai, M. Lalmas, and A.P. de Vries. The overlap problem in content-oriented XML retrieval evaluation. In *27th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, Sheffield, UK, 2004.

[13] G. Geunbae Lee, S. Lee, H. Jung, B-H Cho, C. Lee, B-K Kwak, J. Cha, D. Kim, J. An, J. Seo, H. Kim, and K. Kim. SiteQ: Engineering high performance QA system using lexico-semantic pattern matching and shallow NLP. In *10th Text REtrieval Conference*, Gaithersburg, Maryland, November 2001.

[14] M. Light, G. Mann, E. Riloff, and E. Breck. Analyses for elucidating current question answering technology. *Journal of Natural Language Engineering*, 7(4), 2001.

[15] Das Madirakshi, R. Manmatha, and Edward M. Riseman. Indexing flower patent images using domain knowledge. *IEEE Intelligent Systems*, 14(5), 1999.

[16] Christof Monz. *From Document Retrieval to Question Answering*. Ph. D. dissertation, Universiteit Van Amsterdam, December 2003.

[17] Apostol Natsev, Yuan-Chi Chang, John R. Smith, Chung-Sheng Li, and Jeffrey Scott Vitter. Supporting incremental join queries on ranked inputs. In *Proceedings of the 27th International Conference on Very Large Data Bases*, pages 281–290, Rome, Italy, 2001.

[18] Berthier Ribeiro-Neto, Edleno S. Moura, and Marden S. Neubert. Efficient distributed algorithms to build inverted files. In *22nd Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 105–112, Berkeley, California, August 1999.

[19] Tomek Strzalkowski and Sanda Harabagiu, editors. *Advances in Open Domain Question Answering*. Kluwer Academic Press, 2004.

[20] Stefanie Tellex, Boris Katz, Jimmy Lin, Aaron Fermandes, and Gregory Marton. Quantitative evaluation of passage retrieval algorithms for question answering. In *26nd Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 41–47, Toronto, Canada, July 2003.

[21] Ellen M. Voorhees and Dawn Tice. Building a question answering test collection. In *23rd Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 200–207, Athens, August 2000.

[22] Roger Weber and Michael Mlivoncic. Efficient region-based image retreival. In *12th ACM International Conference on Information and Knowledge Management*, pages 69–76, New Orleans, 2003.

[23] Ian H. Witten, Alistair Moffat, and Timothy C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images*. Morgan Kaufmann Publishing, San Francisco, second edition, 1999.