Efficient Query Evaluation using a Two-Level Retrieval Process

Andrei Z. Broder[§], David Carmel[*], Michael Herscovici[*], Aya Soffer[*], Jason Zien[†]
[§]IBM Watson Research Center, 19 Skyline Drive, Hawthorne, NY 10532
[*]IBM Research Lab in Haifa, MATAM, Haifa 31905, ISRAEL
[†]IBM Almaden Research Center, 650 Harry Road, San Jose, CA 95120

ABSTRACT

We present an efficient query evaluation method based on a two level approach: at the first level, our method iterates in parallel over query term postings and identifies candidate documents using an *approximate evaluation* taking into account only partial information on term occurrences and no query independent factors; at the second level, promising candidates are *fully evaluated* and their exact scores are computed. The efficiency of the evaluation process can be improved significantly using dynamic pruning techniques with very little cost in effectiveness. The amount of pruning can be controlled by the user as a function of time allocated for query evaluation. Experimentally, using the TREC Web Track data, we have determined that our algorithm significantly reduces the total number of full evaluations by more than 90%, almost *without any loss* in precision or recall.

At the heart of our approach there is an efficient implementation of a new Boolean construct called **WAND** or *Weak AND* that might be of independent interest.

Categories and Subject Descriptors

H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval

General Terms

Algorithms, Performance

Keywords

Efficient query evaluation, Document-at-a-time, **WAND**

1. INTRODUCTION

Fast and precise text search engines are widely used in both enterprise and Web environments. While the amount of searchable data is constantly increasing, users have come to expect sub-second response time and accurate search results regardless of the complexity of the query and the size of the data set. Thus, system runtime performance is an increasingly important concern.

We propose a two level approach as a method for decreasing runtime latency: first, we run a fast, approximate evaluation on candidate documents, and then we do a full, slower evaluation limited to promising candidates. Compared to the naive approach of fully evaluating every document that contains at least one of the query terms, our method achieves on average a 92.6% reduction in the number of full evaluations for short queries, and a 95.2% reduction for long queries, *without any loss* in precision or recall. If a slight loss is tolerable, the reduction is 98.2% for short queries, and 98.9% for long queries.

Such a two level evaluation approach is commonly used in other application areas to enhance performance: databases sometimes use Bloom filters [4] to construct a preliminary join followed by an exact evaluation [17]; speech recognition systems use a fast but approximate acoustic match to decrease the number of extensions to be analyzed by a detailed match [2]; and program committee often use a first "rough cut" to reduce the number of papers to be fully discussed in the committee.

1.1 Prior work

Turtle and Flood [20] classify evaluation strategies into two main classes:

- *Term-at-a-time* (*TAAT*) strategies process query terms one by one and accumulate partial document scores as the contribution of each query term is computed.
- *Document-at-a-time* (*DAAT*) strategies evaluate the contributions of every query term with respect to a single document before moving to the next document.

TAAT strategies are more commonly used in traditional IR systems. For small corpus sizes, implementations which use *TAAT* strategies are elegant and perform well. For large corpora, *DAAT* strategies have two advantages: (1) *DAAT* implementations require a smaller run-time memory footprint because a per document intermediate score does not need to be maintained, and (2) they exploit I/O parallelism more effectively by traversing postings lists on different disk drives simultaneously.

Both *TAAT* and *DAAT* strategies can be optimized significantly by compromising on the requirement that all document scores are complete and accurate. Optimization strategies have been studied extensively in the information retrieval literature. For a comprehensive overview of optimization techniques see Turtle and Flood [20]. For *TAAT* strate-

CIKM'03, November 3–8, 2003, New Orleans, Louisiana, USA.

gies, the basic idea behind such optimization techniques is to process query terms in some order that lets the system identify the top n scoring documents without processing all query terms [6, 12, 18, 16, 21]. An important optimization technique for *DAAT* strategies is termed *max-score* by Turtle and Flood. Given a recall parameter n, it operates by keeping track of the top n scoring documents seen so far. Evaluation of a particular document is terminated as soon as it is clear that this document will not place in the top n.

Turtle and Flood demonstrated experimentally that in an environment where it is not possible to store intermediate scores in main memory, optimized *DAAT* strategies outperform optimized *TAAT* strategies. In contrast, Kaszkiel and Zobbel [14] showed that for long queries, the *max-score* strategy is much more costly than optimized *TAAT* strategies, due to the required sorting of term postings at each stage of evaluation; a process that heavily depends on the number of query terms.

The results mentioned above are based on *context-free* queries, i.e. queries that can be evaluated term by term independently, ignoring *context-sensitive* queries for which the relation among terms is crucial. Such queries might contain Boolean operators, proximity operators, and others. As mentioned in [20], most of the context-sensitive queries can be evaluated much more efficiently using *DAAT* strategies. For example, identifying whether two query terms are found in proximity to each other in a given document can easily be determined by verifying the occurrence of both terms together in that document. In contrast, a *TAAT* strategy must keep the occurrences of the first term within the document in the intermediate results, in order to verify whether the second term satisfies the proximity constraint.

Web search engines which must handle context-sensitive queries over very large collections indeed employ *DAAT* strategies; see, for instance, [8] that describes AltaVista [1], and [5] that describes Google [11].

1.2 Our approach

In this paper we describe a novel algorithm suited for *DAAT* strategies that evaluates queries using two levels of granularity. The algorithm iterates in parallel over query term postings and identifies candidate documents using a *preliminary evaluation* taking into account only partial information on term occurrences and no query independent factors. Once a candidate document is identified, it is *fully evaluated* and its exact score is computed. Furthermore, as in the standard *DAAT* approach, our algorithm iterates in parallel over query term postings but the nature of the preliminary evaluation is such that it is possible to skip quickly over large portions of the posting lists. If the result of this "fast and rough" evaluation is above a certain threshold, varied dynamically during the execution, then a *full evaluation* is performed and the exact score is computed.

For large systems the full evaluation is an expensive task that depends on query dependent factors such as term occurrences within the document, as well as query independent properties such as document length, topological score based on link analysis (for HTML pages), etc. Some of these factors might have to be retrieved via an I/O operation but even if all these properties can be computed efficiently, there is still a substantial cost to combine them for a final score. Therefore the intention of the two-level process is to minimize the number of full evaluations as much as possible.

Our approach allows both *safe optimization* and *approximate optimization*. (This terminology has been introduced in [20]). For safe optimization, the two-level strategy makes no false-negative errors and thus it is guaranteed to return the top documents in the correct order and with accurate scores. For an approximate optimization, dynamic pruning techniques are used such that fewer documents pass the preliminary evaluation step, that is, we allow some false-negative errors at the risk of missing some candidate documents whose accurate scores would have placed them in the returned document set. The amount of pruning can be controlled by the user as a function of time allocated for query evaluation. We show experimentally that the efficiency of the evaluation process can be improved significantly using dynamic pruning with very little cost in effectiveness. (See section 4.)

1.3 Experimental results

Our experiments were done on the Juru [9] system, a Java search engine developed at the IBM Research Lab in Haifa. Juru was used to index the WT10G collection of the TREC WebTrack [13] and we measured precision and efficiency over a set of 50 WebTrack topics. Precision was measured by precision at 10 (P@10) and mean average precision (MAP) [22]. The main method for estimating the efficiency of our approach is counting the number of full evaluations required to return a certain number of top results. This measure has the advantage of being independent of both the software and the hardware environment, as well as the quality of the implementation. We also measure query search time and it shows very high correlation with the full number of evaluations. Thus, we consider the number of full evaluations measure a good proxy for wall clock time.

For approximate optimization strategies (when we can have false negatives) we measure two parameters: First we measure the performance gain, as before. Second, we measure the change in recall and precision by looking at the distance between the set of original results and the set of results produced by dynamic pruning. Our main results, described in Section 4, demonstrate that our approach significantly reduces the total number of full evaluations while precision is only slightly decreased.

2. THE TWO-LEVEL EVALUATION PROCESS

In this section we provide complete details of the two-level evaluation process. Recall that the algorithm iterates over the list of documents identifying candidates using an approximate score. Once such a candidate is identified it is fully evaluated. The algorithm keeps track of the top scoring documents seen so far, under full evaluation. A valid candidate will be a document whose approximate score is greater than the minimal score of all documents in the top scoring set so far.

2.1 Basic assumptions

Our model assumes a traditional inverted index for IR systems in which every index term is associated with a posting list. This list contains an entry for each document in the collection that contains the index term. The entry consists of the document's unique positive identifier, DID, as well as any other information required by the system's scoring

model such as number of occurrences of the term in the document, offsets of occurrences, etc. Posting lists are ordered in increasing order of the document identifiers.

From a programming point of view, in order to support complex queries over such an inverted index it is convenient to take an object oriented approach [7, 10]. Each index term is associated with a basic iterator object (a "stream reader" object in the terminology of [7]) capable of sequentially iterating over its posting list. The iterator can additionally skip to a given entry in the posting list. In particular, it provides a method next(id) which returns the first posting element for which DID $\geq id$. If there is no such document, the term iterator returns a special posting element with an identifier $LastID$ which is larger than all existing DIDs in the index.

Boolean and other operators (or predicates) are associated with compound iterators, built from the basic iterators. For instance, the next method for the operator A **OR** B is defined by

$$(A \,\mathbf{OR}\, B).\text{next}(id) = \min(A.\text{next}(id), B.\text{next}(id)).$$

2.2 The WAND operator

At the heart of our approach, there is a new Boolean predicate called **WAND** standing for Weak **AND**, or Weighted **AND**. **WAND** takes as arguments a list of Boolean variables $X_1, X_2, \ldots, X_k$, a list of associated positive *weights*, $w_1, w_2, \ldots, w_k$, and a threshold θ.

By definition, **WAND**$(X_1, w_1, \ldots X_k, w_k, \theta)$ is true iff

$$\sum_{1 \leq i \leq k} x_i w_i \geq \theta, \tag{1}$$

where x_i is the indicator variable for X_i, that is

$$x_i = \begin{cases} 1, & \text{if } X_i \text{ is true} \\ 0, & \text{otherwise.} \end{cases}$$

Observe that **WAND** can be used to implement **AND** and **OR** via

$$\mathbf{AND}(X_1, X_2, \ldots X_k) \equiv \mathbf{WAND}(X_1, 1, X_2, 1, \ldots X_k, 1, k),$$

and

$$\mathbf{OR}(X_1, X_2, \ldots X_k) \equiv \mathbf{WAND}(X_1, 1, X_2, 1, \ldots X_k, 1, 1).$$

Thus by varying the threshold, **WAND** can move from being close to an **OR** to being close to an **AND**, which justifies its name as "weak and". This continuity is comparable in nature to the *p-norm model* [19] that intermediates between Boolean and vector-space processing models. In this model the relationships between query terms described by the Boolean operators can be varied in importance. At one extreme, Boolean relationships are ignored and the model is equivalent to the cosine measure. At the other extreme, the model is equivalent to the fuzzy set model in which documents must exactly match the Boolean expression.

We also note that **WAND** can be generalized by replacing condition (1) by requiring an arbitrary monotonically increasing function of the x_i's to be above the threshold, or, in particular, by requiring an arbitrary monotone Boolean formula to be True. (A monotone Boolean formula is such that changing a variable from F to T can never invalidate the formula. It is always expressible as an **OR** of **AND**s with no negated literals.)

2.3 Scoring

The final score of a document involves a textual score which is based on the document textual similarity to the query, as well as other query independent factors such as connectivity for web pages, citation count for scientific papers, inventory for e-commerce items, etc. To simplify the exposition, we shall assume for the time being that there are no such query independent factors and will explain how to deal with them in practice when we discuss our experimental results.

We assume an additive scoring model, that is, the textual score of each document is determined by summing the contribution of all query terms belonging to the document. Thus, the textual score of a document d for query q is:

$$\text{Score}(d, q) = \sum_{t \in q \cap d} \alpha_t w(t, d) \tag{2}$$

For example, for the $tf \times idf$ scoring model used by many IR systems, α_t is a function of the number of occurrences of t in the query, multiplied by the inverse document frequency (idf) of t in the index and $w(t, d)$ is a function of the term frequency (tf) of t in d, divided by the document length $|d|$.

In addition we assume that each term is associated with an upper bound on its maximal contribution to any document score, UB_t such that

$$UB_t \geq \alpha_t \max(w(t, d_1), w(t, d_2), \ldots).$$

By summing the upper bounds of all query terms appearing in a document, we can determine an upper bound on the document's query-dependent score.

$$UB(d, q) = \sum_{t \in q \cap d} UB_t \geq Score(d, q). \tag{3}$$

Note that query terms can be simple terms, i.e., terms for which a static posting list is stored in the index, or complex terms such as phrases, for which the posting list is created dynamically during query evaluation. The model does not distinguish between simple and complex terms; each term must provide an upper bound, and for implementation purposes each term must provide a posting iterator.

Given this setup our preliminary scoring consists of evaluating for each document d

$$\mathbf{WAND}(X_1, UB_1, X_2, UB_2, \ldots, X_k, UB_k, \theta),$$

where X_i is an indicator variable for the presence of query term i in document d and the threshold θ is varied during the algorithm as explained below. If **WAND** evaluates to true, then the document d undergoes a full evaluation.

The threshold θ is set dynamically by the algorithm based on the minimum score m among the top n results found so far, where n is the number of requested documents. The larger the threshold, the more documents will be skipped and thus we will need to compute full scores for fewer documents. It is easy to see that if the term upper bounds are accurate, then the final score of a document is no greater than its preliminary upper bound, and therefore all documents skipped by WAND with $\theta = m$ would not be placed in the top scoring document set by any other alternative scheme that uses the same additive scoring model.

However, as explained later, (a) we might have only approximate upper bounds for the contribution of each term, (b) the score might involve query independent factors, and

(c) we might want to use a higher threshold in order to execute fewer full evaluations. Thus in practice we will set $\theta = F \cdot m$ where F is a *threshold factor* chosen to balance the positive and negative errors for the collection.

2.4 Implementing the WAND iterator

We now describe how to iteratively find candidates for full evaluation using the **WAND** predicate. To this end, we must build a WAND iterator, that is a procedure that can quickly find the documents that satisfy the predicate.

The WAND iterator is initialized by calling the init() method depicted in pseudo-code in Figure 1. The method receives as input the array of query terms. It sets the current document to be considered (curDoc) to zero and for each query term, t, it initializes its current posting *posting*[t] to be the first posting element in its posting list.

```
1. Function init(queryTerms)
2.     terms ← queryTerms
3.     curDoc ← 0
4.     for each t ∈ terms
5.         posting[t] ← t.iterator.next(0)
```

Figure 1: The init() method of the WAND iterator

After calling the init() method, the algorithm repeatedly calls WAND's next() method to get the next candidate for full evaluation. The next() method takes as input a threshold θ and returns the next document whose approximate score is larger than θ. Documents whose approximate score is lower than the threshold are skipped.

Figure 2 contains pseudo-code of the next() method.

The WAND iterator maintains two invariants during its execution:

1. All documents with DID $\leq curDoc$ have already been considered as candidates.

2. For any term t, any document containing t, with DID $<$ posting[t].DID, has already been considered as a candidate.

Note that the init() method establishes these invariants.

The WAND iterator repeatedly advances the individual term iterators until it finds a candidate document to return. This could be performed in a naive manner by advancing all iterators together to their next document, approximating the scores of candidate documents in DID order, and comparing to the threshold. This method would, however, be very inefficient and would require several superfluous disk I/O's and computation. Our algorithm is optimized to minimize the number of next() operations and the number of approximate evaluations. It first sorts the query terms in increasing order of the DID's of their current postings. Next, it computes a *pivot term* – the first term in this order for which the accumulated sum of upper bounds of all terms preceding it, including it, exceeds the given threshold. The pivot DID is the smallest DID that might be a candidate. If there is no such term (meaning the sum of all term upper bounds is less than the threshold) the iterator stops and returns the constant $NoMoreDocs$.

```
1. Function next(θ)
2.     repeat
3.         /* Sort the terms in non decreasing order of
           DID */
4.         sort(terms, posting)
5.         /* Find pivot term - the first one with accumulated
           UB ≥ θ */
6.         pTerm ← findPivotTerm(terms, θ)
7.         if (pTerm = null) return (NoMoreDocs)
8.         pivot ← posting[pTerm].DID
9.         if (pivot = lastID) return (NoMoreDocs)
10.        if (pivot ≤ curDoc)
11.            /* pivot has already been considered, advance
               one of the preceding terms */
12.            aterm ← pickTerm(terms[0..pTerm])
13.            posting[aterm] ← aterm.iterator.next(curDoc+1)
14.        else /* pivot > curDoc */
15.            if (posting[0].DID = pivot)
16.                /* Success, all terms preceding pTerm belong
                   to the pivot */
17.                curDoc ← pivot
18.                return (curDoc, posting)
19.            else
20.                /* not enough mass yet on pivot, advance
                   one of the preceding terms */
21.                aterm ← pickTerm(terms[0..pTerm])
22.                posting[aterm] ← aterm.iterator.next(pivot)
23.        end repeat
```

Figure 2: The next() method of the WAND iterator

The pivot variable is set to the DID corresponding to the current posting of the pivot term. If the pivot is less or equal to the DID of the last document considered (curDoc), WAND picks a term preceding the pivot term and advances its iterator past curDoc, the reason being that all documents preceding curDoc have already been considered (by Invariant 1) and therefore the system should next consider a document with a larger DID. Note that this move preserves Invariant 2. (NB: In fact in Line 10 the case Pivot $<$ curDoc cannot happen, but this requires an extra bit of proof.)

If the pivot is greater than curDoc, we need to check whether indeed the sum of contributions to the pivot document is greater than the threshold. There are two cases: if the current posting DID of all terms preceding the pivot term is equal to the pivot document, then the pivot document contains a set of query terms with an accumulated upper bound larger than the threshold and hence next() sets curDoc to the pivot, and returns this document as a candidate for full evaluation. Otherwise, the pivot document might or might not contain all the preceding terms, that is, it might or might not have enough contributions, hence WAND picks one of these terms and advances its iterator to a location $\geq$ the pivot location.

Note that the next() method maintains the invariant that all the documents with DID $\leq$ curDoc have already been considered as candidates (Invariant 1). It is not possible for another document whose DID is smaller than that of the pivot to be a valid candidate since the pivot term by definition is the first term in the DID order for which the accumulated upper bound exceeds the threshold. Hence, all documents with a smaller DID than that of the pivot can only contain terms which precede the pivot term, and

thus the upper bound on their score is strictly less than the threshold. It follows that next() maintains the invariant since curDoc is only advanced to the pivot document in the cases of success, i.e., finding a new valid candidate who is the first in the order.

The next() method invokes three helper methods, sort(), findPivotTerm() and pickTerm(). The first helper, sort(), sorts the terms in non decreasing order of their current DID. Note that there is no need to fully sort the terms at any stage since only one term advances its iterator between consecutive calls to sort; hence, by using an appropriate data structure, the sorted order is maintained by modifying the position of only one term. The second helper, findPivotTerm(), returns the first term in the sorted order for which the accumulated upper bounds of all terms preceding it, including it, exceed the given threshold.

The third helper, pickTerm(), receives as input a set of terms and selects the term whose iterator is to be advanced. An optimal selection strategy will select the term which will produce the largest expected skip. Advancing term iterators as much as possible reduces the number of documents to consider and hence the number of postings to retrieve. Note however that this policy has no effect on the set of documents that are fully evaluated. Any document whose score upper bound is larger than the threshold, will be evaluated under any strategy. Thus, while a good pickTerm policy may improve performance, it cannot affect precision. In our current implementation pickTerm() selects the term with the maximal idf, assuming the rarest term will produce the largest skip. Identifying an optimal selection strategy is outside the scope of this paper. A recent on-going research on this issue is described in [3].

2.5 Setting the WAND Threshold

Assume that we wish to retrieve the top n scoring documents for a given query. The algorithm will maintain a heap of size n to keep track of the top n results. After calling the init() method of the WAND iterator, the algorithm calls the next() method to receive a new candidate. When a new candidate is returned by the WAND iterator, this document is fully evaluated using the system's scoring model resulting in the precise score for this document. If the heap is not full the candidate is inserted into the heap. If the heap is full and the new score is larger than the minimum score in the heap, the new document is inserted into the heap, replacing the one with the minimum score.

The threshold value that is passed to the WAND iterator is set based on the minimum score of all documents currently in the heap. Recall that this threshold determines the lower bound that must be exceeded for a document to be considered as candidate and to be passed to the full evaluation step.

The initial threshold is set based on the query type. For an OR query, or for a free-text query, the initial threshold is set to zero. The approximate score of any document that contains at least one of the query terms would exceed this threshold and would thus be returned as a candidate. Once the heap is full and a more realistic threshold is set, we only fully evaluate documents that have enough terms to yield a high score. For an AND query, the initial threshold can be set to the sum of all term upper bounds. Only documents containing all query terms would have a high enough approximate score to be considered candidates.

The initial threshold can also be used to handle mandatory terms (those preceded by a '+'). The upper bound for such terms can be set to some huge value, H, which is much larger than the sum of all the other terms upper bounds. By setting the initial threshold to H, only documents containing the mandatory term will be returned as candidates. If the query contains k mandatory terms, the initial threshold should be set to $k \cdot H$.

So far we have only described methods in which we are guaranteed to return accurate results for the query (safe evaluation). However, the threshold can additionally be used to expedite the evaluation process by being more opportunistic in terms of selecting candidate documents for full evaluation. In this case, the threshold would be set to a value larger than the minimum score in the heap. By increasing the threshold, the algorithm can dynamically prune documents during the approximation step and thus fully evaluate less overall candidates but with higher potential. The cost of dynamic pruning is the risk of missing some high scoring documents and thus the results are not guaranteed to be accurate. However, in many cases this can be a very effective technique. For example, systems that govern the maximum time spent on a given query can increase the threshold when the time limit is approaching thus enforcing larger skips and fully evaluating only documents that are very likely to make the final result list. In Section 4 we show experimentally how dynamic pruning affects the efficiency as well as the effectiveness of query evaluation using this technique.

3. COMPUTING TERM UPPER BOUNDS

The WAND iterator requires that each query term t be associated with an upper bound, UB_t, on its contribution to any document score. Recall that the upper bound on the document score is computed by summing the upper bounds of all terms that the document contains. It is therefore clear that if the term upper bounds are accurate, i.e., $\forall t, UB_t \geq \alpha_t \max_d w(t,d)$, then the upper bound on the score of a document is also accurate i.e, it is greater than its final score. In this case, it guaranteed that assuming the algorithm sets the threshold at any stage to the minimum document score seen so far, the two-level process will return correct ranking and accurate document scores.

It is easy to find a true upper bound for simple terms. Such terms are directly associated with a posting list that is explicitly stored in the index. We first traverse the term's posting list and for each entry compute the contribution of this term to the score of the document corresponding to this entry. We then set the upper bound to the maximum contribution over all posting elements. This upper bound is stored in the index as one of the term's properties. The problem with this approach is that a term upper bound might be set to an extremely high value when the term is extremely frequent in one particular document but infrequent in all other documents. As a result of such an abnormality, the approximate score of most documents containing this term will be much higher than the true score of the document. These documents will pass the approximate evaluation threshold and will thus be passed to the full evaluation step where they will be rejected. We call such cases false positive errors, since the approximation incorrectly evaluated the document as a candidate for the top scoring set. Such errors impact efficiency since these documents are fully evaluated but not inserted into the heap. It follows that special attention needs

to be given to upper bound estimation even for simple terms. Furthermore, for complex query terms such as phrases or proximity pairs, term upper bounds must be estimated since their posting lists are created dynamically during query evaluation.

In the following we describe an alternative method for upper bound estimation of simple terms as well as schemes for estimating upper bounds for complex terms. For simple terms, we approximate the upper bound for a term t to be $UB_t = C \cdot \alpha_t$. Recall that α_t is determined by the term's idf and the term's frequency in the query. $C > 1$ is a constant which is uniformly used for all terms. This estimate ignores other factors that usually affect the contribution of a specific term to the document's scores. These include term frequency in the document, the context of the occurrence (e.g., in the document title), document length and more.

The benefit of this estimate is its simplicity. The tradeoff is that the computed upper bound of a candidate document can now be lower than the document's true score resulting in false negative errors. Such errors may result in incorrect final rankings since top scoring documents may not pass the preliminary evaluation step and will thus not be fully evaluated. Note however, that false negative errors can only occur once the heap is full and if the threshold is set to a high value.

The parameter C can be fine tuned for a given collection to balance between false positive errors and false negative errors. The larger C, the more false positive errors are expected and thus system efficiency is harmed. Decreasing C will cause more false negative errors and thus hurt the effectiveness of the system. In Section 4 we show experimentally that C can be set to a relatively small value before the system's effectiveness is impaired.

3.1 Estimating the upper bound for complex terms

As described above we estimate the upper bound for a query term based on its inverse document frequency (*idf*). The *idf* of simple terms can easily be determined by gathering and storing the true *df* statistics at indexing time. The *idf* of complex terms that are not explicitly stored as such in the index must be estimated since their posting lists are created dynamically during query evaluation. In this subsection we describe how to estimate the *idf* of two types of complex terms. These methods can be extended to other types of complex terms.

• **Phrases:** A phrase is a sequence of query terms usually wrapped in quotes. A document satisfies this query only if it contains all of the terms in the phrase in the same order as they appear in the phrase query. Note that in order to support dynamic phrase evaluation the postings of individual terms must include the offsets of the terms within the document. Moreover, phrase evaluation necessitates storing stop-words in the index.

For each phrase we build an iterator outside WAND. Inside WAND, since phrases are usually rare we treat phrases as "must appear" terms, that is, only documents containing the query phrases are retrieved. Recall that our method handles mandatory terms by setting their upper bound to a huge value H, regardless of their *idf*. In addition, the threshold is also initialized to H. Thus, only candidate documents containing the phrase will pass to the detailed evaluation step.

• **Lexical affinities:** Lexical affinities (LAs) are terms found in close proximity to each other, in a window of small size [15]. The posting iterator of an LA term receives as input the posting iterators of both LA terms and returns only documents containing both terms in close proximity. In order to estimate the document frequency of an LA (t_1, t_2), we make use of the fact that the posting list of the LA is a sub-sequence of the posting lists of its individual terms. We count the number of appearances of the LA in the partial posting lists of its terms traversed so far and extrapolate to the entire posting lists.

More specifically, the document frequency of the LA is initialized to $df_0(LA) = \min(df(t_1), df(t_2))$, and is updated repeatedly after traversing an additional k documents. Let $p(t_i)$ be the posting list of term t_i and $p'(t_i)$ be its partial posting list traversed so far. Let $\#(LA \mid p'(t_i))$ be the number of documents containing the LA in $p'(t_i)$. We can estimate the number of documents containing the LA in the entire posting list of t_i by the extrapolation

$$\#(LA \mid p(t_i)) = \frac{\#(LA \mid p'(t_i))}{|p'(t_i)|} |p(t_i)| .$$

It follows that the update rule for the document frequency of the LA at stage n is:

$$df_n(LA) = \min\left[df_{n-1}(LA), \frac{\#(LA \mid p(t_1)) + \#(LA \mid p(t_2))}{2}\right]$$

The rate of convergence depends on the length of the term posting lists. In the experiments we conducted (not reported in this work due to lack of space) we show that the document frequency estimation of LAs quickly converges after only a few iterations.

4. EXPERIMENTAL RESULTS

In this section we report results from experiments which we conducted to evaluate the proposed two-level query evaluation process. For our experiments we used Juru [9], a Java search engine, developed at the IBM Research Lab in Haifa. We indexed the WT10G collection of the TREC WebTrack [13] which contains 10GB of data consisting of 1.69 million HTML pages. We experimented with both short and long queries. The queries were constructed from topics 501-550 of the WebTrack collection. We used the topic title for short query construction (average 2.46 words per query), and the title concatenated with the topic description for long query construction (average 7.0 words per query). In addition we experimented with the size of the result set (the heap size). The larger the heap, more evaluations are required to obtain the result set.

We experimented with the independent parameter C, the constant which multiplies the sum of the of the query term upper bounds to obtain the document score upper bound. Recall that we compare the threshold parameter passed to the **WAND** iterator with the documents' score upper bound. Documents are fully evaluated only if their upper bound is greater than the given threshold. C, therefore, governs the tradeoff between performance and precision; the smaller C is, less documents will be fully evaluated, at the cost of lower precision, and vice versa. For practical reasons, instead of varying C, we fixed its value and varied the value of the threshold factor F that multiplies the true threshold passed

to the **WAND** iterator. The factor C is in inverse relation to F, therefore varying F is equivalent to varying C with the opposite effect. That is, large values of F result in fewer full evaluations and in an expected loss in precision. Setting these values in Equation 1, a document will be returned by **WAND**$(X_1, C\alpha_1, \ldots X_k, C\alpha_k, F\theta)$ if and only if $C\sum_{1\leq i\leq k} x_i\alpha_i \geq F\theta$. When setting F to zero the threshold passed to **WAND** is zero and thus all documents that contain at least one of the query terms are considered candidates and fully evaluated. When setting F to an infinite value, the algorithm will only fully evaluate documents until the heap is full (until $\theta > 0$). The remainder of the documents will not pass the threshold since $F\theta$ will be greater than the sum of all query term upper bounds.

We measured the following parameters for varying values of the threshold factor:

- Average time per query.
- Average number of full evaluations per query. This is the dominant parameter that affects search performance. Clearly, the more full evaluations, the slower the system.
- Search precision as measured by precision at 10 (P@10) and mean average precision (MAP) [22].
- The difference between the search result set obtained from a run with no false-negative errors (the basic run) and the result set obtained from runs with negative errors (pruned runs).

  Note that documents receive identical scores in both runs, since the full evaluator is common and it assigns the final score; hence the relative order of common documents in the basic set B and the pruned set P is maintained. Therefore if each run returns k documents, the topmost j documents returned by the pruned run, for some $j \leq k$, will be in the basic set and in the same relative order.

  We measure the difference between the two result sets in two ways. First we measure the relative difference given by the formula

$$\frac{|B \setminus P|}{|B|} = \frac{k-j}{k} \tag{4}$$

  Second, since not all documents are equally important, we measure the difference between the two result sets using MRR (mean reciprocal rank) weighting. Any document that is in the basic set, B, in position i in the order, but is not a member of the pruned set, P, contributes $1/i$ to the MRR distance. The idea is that missing documents in the pruned set contribute to the distance in inverse relation to their position in the order. The MRR distance is normalized by the MRR weight of the entire set. Thus

$$MRR(B,P) = \frac{\sum_{i=1,d_i\in B-P}^{k} 1/i}{\sum_{i=1}^{k} 1/i} \tag{5}$$

4.1 Effectiveness and efficiency

In the first experiment we measured the efficiency of the evaluation process by measuring the average query time and the number of full evaluations, as a function of the threshold parameter F. By setting F to zero all documents which contain at least one query term are considered candidates and are fully evaluated. We use this as a base run and observe that on average 335,500 documents are evaluated per long query, while 135,000 documents are evaluated per short query.

Figure 3 shows the average query time for short queries as a function of the threshold factor. While the average query time of the base run is extremely high, it quickly converges to sub-second time even for small F values. Note that the base run is an extreme case were no pruning is performed. We can actually set the threshold to a higher value before any negative errors occur. Based on our experiments, a threshold of approximately 1.0 results in significant reduction of the average query time to a sub-second time with no effect on the result list.

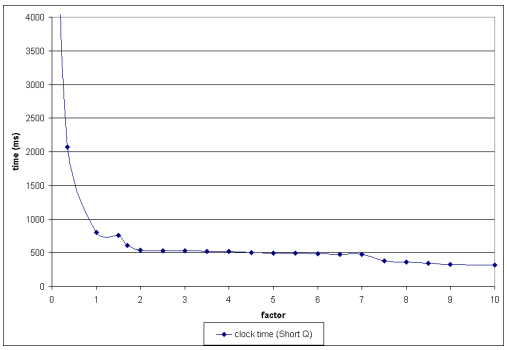

Figure 3: The average query time for short queries as a function of the threshold factor (heap size H=1000).

Figure 4 shows the number of full evaluations as a function of the threshold factor F, for long and for short queries, and for a heap size of 100 and 1000. We can see that for all runs, as F increases, the number of evaluations quickly converges to the number of required documents (the heap size).

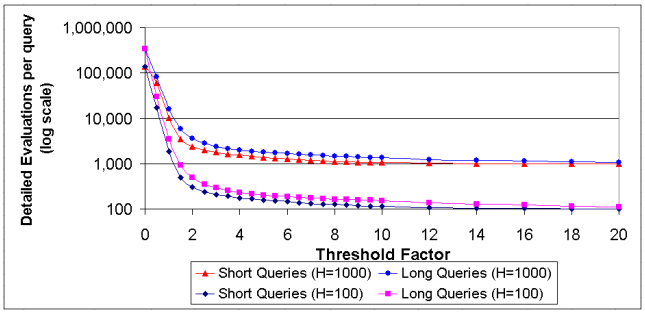

Figure 4: The number of full evaluations as a function of the threshold factor. (H = the heap size.)

Figure 5 shows the difference between the pruned results and the base results for the same runs as measured by the relative difference measure (Equation 4) and the MRR distance measure (Equation 5). For small values of F the distance is zero since there are no false negative errors. Increasing F increases the number of false negative errors hence the distance increases.

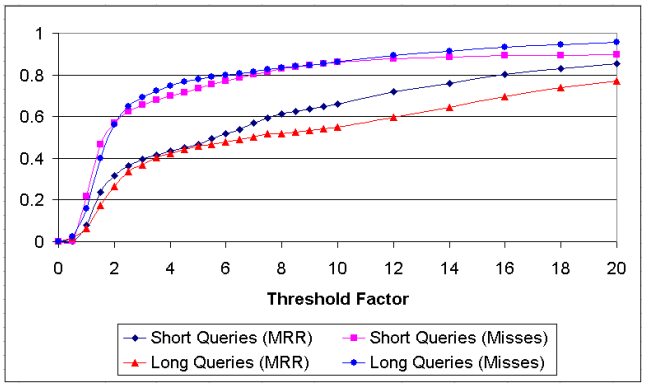

Figure 5: Relative difference (misses) and MRR distance as a function of the threshold factor.

Figure 6 shows the precision of the same runs, as measured by P@10 and MAP, for short and long queries with a heap size of 1000. We can see that while MAP decreases as we prune more (as expected), the change in P@10 is negligible.

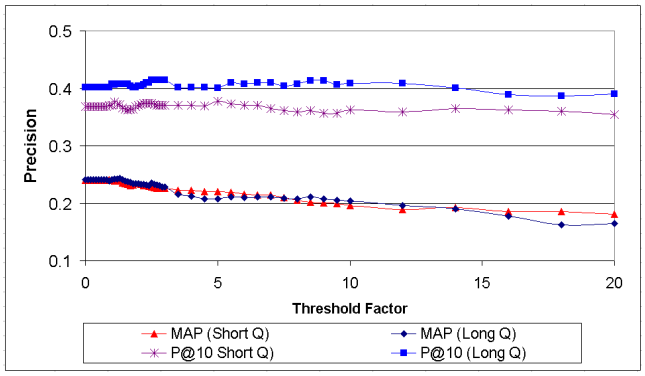

Figure 6: Precision as a function of the threshold factor.

The reason for high precision in the top results set, even under aggressive pruning, is explained by the fact that a high threshold in essence makes **WAND** behave like an **AND** returning only documents that contain all query terms. These documents are then fully evaluated and most likely receive a high score. Since the scores are not affected by the two-level process, and since these documents are indeed relevant and receive a high score in any case, P@10 is not affected. On the other hand, MAP which also takes into account recall is hurt because of the many misses.

This surprising result led us to hypothesize that by explicitly evaluating only documents containing all query terms the system can achieve high precision in the top result set. **WAND** can easily be instructed to return only such documents by passing it a threshold value that is equal to the sum of all query term upper bounds. We call this experiment *AllTerms* run. While this approach indeed proves itself in terms of P@10, the recall and therefore the MAP decreases since too few documents are considered for many queries. A modified strategy, named *TwoPass*, allows a second pass over the term postings in case the first "aggressive" pass did not return enough results. Specifically, the threshold is first set to the sum of all term upper bounds; if the number of accumulated documents is less than the required number of results, the threshold is reduced and set to the largest upper bound of all query terms that occur at least once in the corpus, and the evaluation process is re-invoked.

Table 1 shows the results of **WAND** with some different threshold factors, compared to the *AllTerms* and the *TwoPass* runs. For $F = 0$, **WAND** returns all documents that contains at least one of the query terms. For this run since there are no false negative errors the precision is maximal. For $F = 1.0$, the number of full evaluations is decreased by a factor of 20 for long queries and by a factor of 10 for short queries, still without any false negative errors hence with no penalty in precision. For $F = 2.0$ the number of evaluations is further decreased by a factor of 4 at the cost of lower precision.

We clearly see that *AllTerms* improves P@10 significantly compared to **WAND**, both for short and for long queries, while MAP decreases significantly. For systems interested only in precision of the top results, ignoring recall, the *AllTerms* strategy is a reasonable and effective choice. The *TwoPass* run achieves remarkable results both for P@10 and MAP[1]. A small cost is incurred in terms of execution time for the second pass but it is negligible in most cases since the term postings are most likely still cached in main memory from the first pass. In any event these results demonstrate the versatility and flexibility of our method in general and the **WAND** iterator in particular. By varying the threshold we can control the "strength" of the operator varying its behavior from an OR to an AND.

	ShortQ			LongQ		
	P@10	MAP	#Eval	P@10	MAP	#Eval
WAND (F=0)	0.368	0.24	136,225	0.402	0.241	335,500
WAND (F=1.0)	0.368	0.24	10,120	0.402	0.241	15,992
WAND (F=2.0)	0.362	0.23	2,383	0.404	0.234	3,599
AllTerms	0.478	0.187	443.6	0.537	0.142	147
TwoPass	0.368	0.249	22,247	0.404	0.246	29,932

Table 1: P@10 and MAP of *AllTerms* and *TwoPass* runs compared to basic WAND.

5. CONCLUSIONS

Our investigation demonstrates that using the document-at-a-time approach and a two level query evaluation method using the **WAND** operator for the first stage, pruning can yield substantial gains in efficiency at no loss in precision and recall. Furthermore, if we are willing to accept some small loss of precision, we can increase the gains even further and/or modify the pruning dynamically to achieve predetermined time bounds.

From a software engineering point of view, the **WAND** operation fits very well in the modern, object-oriented framework for inverted indices, and allows the modularization of ranking in the sense that we can implement a single **WAND** interface that can be reused in multiple circumstances while particularizing only the full evaluation procedure.

[1]To the best of our knowledge, the precision results achieved by the *TwoPass* run for short queries are among the best results ever reported for the WebTrack ad-hock task [13]

Acknowledgments

We thank Adam Darlow for implementation of the algorithms described in this work.

6. REFERENCES

[1] Altavista. `http://www.altavista.com`.

[2] L. Bahl, P. de Souza, P. Gopalakrishnan, D. Nahamoo, and M. Picheny. A fast match for continuous speech recognition using allophonic models. In *Proceedings of IEEE ICASSP*, pages 17–20, March 1992.

[3] K. Beyer, A. Jhingran, B. Lyle, S. Rajagopalan, and E. Shekita. Pivot join: A runtime operator for text search. Submitted for publication, 2003.

[4] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, July 1970.

[5] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. *WWW7/Computer Networks and ISDN Systems*, 30:107–117, April 1998.

[6] C. Buckley and A. F. Lewit. Optimization of inverted vector searches. In *Proceedings of the Eighth International ACM-SIGIR Conference*, pages 97–110, Montreal, Canada, June 1985.

[7] M. Burrows. Object-oriented interface for an index. US Patent 5,809,502, 1998.

[8] M. Burrows. Method for parsing, indexing and searching world-wide-web pages. US Patent 5,864,863, 1999.

[9] D. Carmel, E. Amitay, M. Herscovici, Y. S. Maarek, Y. Petruschka, and A. Soffer. Juru at TREC 10 - Experiments with Index Pruning. In *Proceeding of Tenth Text REtrieval Conference (TREC-10)*. National Institute of Standards and Technology (NIST), 2001.

[10] C. Clarke, G. Cormack, and F. Burkowski. Shortest substring ranking. In *Proceedings of the Fourth Text Retrieval Conference (TREC-4)*. National Institute of Standards and Technology (NIST), November 1995.

[11] Google. `http://www.google.com`.

[12] D. Harman and G. Candela. Retrieving records from a gigabyte of text on a minicomputer using statistical ranking. *Journal of the American Society of Information Science*, 41(8):581–589, 1990.

[13] D. Hawking and N. Craswell. Overview of the TREC-2001 Web Track. In E. M. Voorhees and D. K. Harman, editors, *Proceedings of the Tenth Text Retrieval Conference (TREC-10)*. National Institute of Standards and Technology (NIST), 2001.

[14] M. Kaszkiel and J. Zobel. Term-ordered query evaluation versus document-ordered query evaluation for large document databases. In *Proceedings of the 21st Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 343–344, Melbourne, Austrailia, August 1998.

[15] Y. Maarek and F. Smadja. Full text indexing based on lexical relations: An application: Software libraries. In *Proceedings of the Twelfth International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 198–206, Cambridge, MA, June 1989.

[16] A. Moffat and J. Zobel. Fast ranking in limited space. In *Proceedings of the 10th IEEE International Conference on Data Engineering,*, pages 428–4376, Houston, TX, February 1994.

[17] J. K. Mullin. Optimal semijoins for distributed database systems. *IEEE Transactions on Software Engineering*, 16(5):558, May 1990.

[18] M. Persin. Document filtering for fast ranking. In *Proceedings of the Seventeenth International ACM-SIGIR Conference on Research and Development in Information Retrieval*, pages 339–348, Dublin, Ireland, July 1994.

[19] G. Salton, E. A. Fox, and H. Wu. Extended boolean information retrieval. *Communications of the ACM*, 26(11):1022–1036, 1983.

[20] H. Turtle and J. Flood. Query evaluation: Strategies and optimizations. *Information Processing and Management*, 31(6):831–850, 1995.

[21] A. N. Vo and A. Moffat. Compressed inverted files with reduced decoding overheads. In *Proceedings of the 21st Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 290–297, Melbourne, Australia, August 1998.

[22] E. M. Voorhees and D. K. Harman. Overview of the Tenth Text REtrieval Conference (TREC-10). In *Proceedings of the Tenth Text Retrieval Conference (TREC-10)*. National Institute of Standards and Technology (NIST), 2001.