Optimization Strategies for Complex Queries

Trevor Strohman
strohman@cs.umass.edu
Center for Intelligent Information Retrieval
Department of Computer Science
University of Massachusetts
Amherst, MA 01003

Howard Turtle
turtle@cogitech.com
Cogitech
Jackson Hole, WY 83001

W. Bruce Croft
croft@cs.umass.edu
Center for Intelligent Information Retrieval
Department of Computer Science
University of Massachusetts
Amherst, MA 01003

ABSTRACT

Previous research into the efficiency of text retrieval systems has dealt primarily with methods that consider inverted lists in sequence; these methods are known as term-at-a-time methods. However, the literature for optimizing document-at-a-time systems remains sparse.

We present an improvement to the max_score optimization, which is the most efficient known document-at-a-time scoring method. Like max_score, our technique, called term bounded max_score, is guaranteed to return exactly the same scores and documents as an unoptimized evaluation, which is particularly useful for query model research. We simulated our technique to explore the problem space, then implemented it in Indri, our large scale language modeling search engine. Tests with the GOV2 corpus on title queries show our method to be 23% faster than max_score alone, and 61% faster than our document-at-a-time baseline. Our optimized query times are competitive with conventional term-at-a-time systems on this year's TREC Terabyte task.

Categories and Subject Descriptors: H3.3 Information Storage and Retrieval: Information Search and Retrieval

General Terms: Algorithms, Design, Performance

Keywords: Indexing, Query Processing, Efficiency

1. INTRODUCTION

It might seem that the performance of retrieval systems would not be a major issue anymore, as computer speeds continue to increase. Unfortunately, the document collections that users wish to access continue to grow. Perhaps more importantly, users have been taught by web search engines to expect query results in less than one second, no matter how large the collection or how complex the query.

Many query optimization strategies have been presented in the literature in order to deal with this problem. In general, these methods take a scoring method, such as cosine similarity or language modeling, and develop an approximation of this method that can be computed efficiently. The best of these optimization methods have impressive speedups over baseline systems and have only a small effect on retrieval effectiveness.

The fastest known query processing techniques are term-at-a-time algorithms, some of which use term frequency sorted lists. These methods use accumulators to store partial scores of documents during query evaluation. This works well because these accumulators are small, and by using optimization strategies the number of accumulators can be trimmed to keep memory usage low. However, when term position information is involved these accumulators must be much larger. Furthermore, traditional accumulator trimming techniques do not apply to queries using term position information.

These problems suggest a document-at-a-time evaluation strategy for complex queries. In document-at-a-time evaluation, all inverted lists are considered simultaneously, so there is no need to store proximity information in memory. However, there has been much less research into optimizing document-at-a-time systems than term-at-a-time systems.

Brown [2] and Turtle and Flood [8] present two of the most promising document-at-a-time query processing strategies. In this paper we show that these strategies are in fact complementary; by combining the essence of these methods, we have an optimization method that is significantly faster than the max_score method of Turtle and Flood. However, it manages to maintain the most desirable properties of max_score; most importantly, the results returned by our method are exactly those that would be returned by an unoptimized system. This makes our optimization method particularly useful for query model research.

2. MOTIVATION

Our research is motivated by the Indri retrieval system, a new language modeling search engine developed at the University of Massachusetts Amherst. This system incorporates recent work by Metzler and Croft [6] which combines the language modeling and inference network approaches to information retrieval. By leveraging this work, Indri supports many of the structured query operators from INQUERY [4]. In addition, Indri supports new operators for dealing with document fields, part of speech and named entity tagging, passage retrieval and numeric quantities.

A subset of the Indri query language constructs is shown

Construct	Name	Description
`#odN` $(q_1, ..., q_n)$	Ordered window	A match occurs if the q_i's appear in order with no more than N words between adjacent terms
`#uwN` $(q_1, ..., q_n)$	Unordered window	A match occurs the q_i's appear in any order within a window of N words
`#any:` *field*	Any operator	A match occurs if any field called *field* is found
term.*field*	Field restriction	A match occurs if *term* is found in a field named *field*
`#combine` $(q_1, ..., q_n)$	Combine operator	Combines inference from the q_i's; similar to `#and` from [6]
`#weight` $(w_1 q_1, ..., w_n q_n)$	Weight operator	Combines inference from the q_i's, using the w_i's as weights; similar to #wand from [6].
`#greater` (*fieldn*)	Greater operator	Evaluates to true if the document contains a field *field* with a numeric value greater than n
`#less` (*field* n)	Less operator	Evaluates to true if the document contains a field *field* with a numeric value less than n
`#equal` (*field* n)	Equal operator	Evaluates to true if the document contains a field *field* with a numeric value equal to n
#*operator*[*field*]$(q_i, ..., q_n)$	Context restriction	Evaluates operator *operator* using *field* as the evaluation context; essentially similar to passage retrieval with sentence passages
`#filrej`(c s)	Filter reject	Score the subquery s only if the condition c is not met
`#filreq`(c s)	Filter require	Score the subquery s only if the condition c is met

Figure 1: Complex query operators from Indri

in Figure 1. These operators can be used together to form complex queries, such as:

```
#combine[sentence]( #uw10( #uw10( george w bush )
                           #any:number
                           president ) )
```

which scores all sentences in the corpus that contain the phrase "George W. Bush" followed by a number, followed by "president". This query could be used to find sentences of the form "George W. Bush is the 43rd president of the United States." Quickly finding passages of text that meet a particular form is a critical part of many question answering systems.

The field structure of the query language can also be used for structured data, as in this query:

```
#combine( #uw1( howard turtle ).author
          #uw1( james flood ).author
          #1( query evaluation ).title )
```

which searches for the paper on which much of this work is based.

It is clear that the Indri query language relies heavily on the location of terms within documents, and therefore a document-at-a-time evaluation strategy is appropriate. However, we would still like to be able to evaluate simpler queries quickly, such as these from the 2004 TREC Terabyte task:

```
pearl farming
prostate cancer treatments
```

Our goal with this work is to achieve good performance on these simpler queries without resorting to an entirely different evaluation strategy than we use for complex queries.

3. RELATED WORK

All modern retrieval systems use inverted lists to evaluate queries efficiently [9]. The differences between the optimizations discussed here lie in how these inverted lists are processed. In term-at-a-time systems, the inverted list for each term is considered separately. These systems use table of score accumulators to keep track of partial scores for documents that have been seen. At the end of evaluation, the accumulators are sorted, and the top k documents (where k is a parameter given by the user) are returned to the user. In document-at-a-time systems, the inverted lists are considered simultaneously, much like the classical merge sort algorithm [5]. In this case, there are no partial scores; this allows the system to maintain a list of the top k scores it has seen so far.

One of the earliest papers on modern query optimization comes from Buckley [3]. In this approach, terms are evaluated one at a time, from the least frequent term to the most frequent term. At some point during query evaluation, it may be possible to show that it is not necessary to consider any more terms, as the document at rank $k + 1$ cannot surpass the score of the document at rank k.

This approach, and approaches based on it, have the advantage that some inverted lists will not need to be read from disk. As the gap between disk access speed and processing speed continues to increase, this is an attractive feature. However, this approach has trouble with duplicate (or nearly duplicate) documents in the collection. If two documents at ranks k and $k + 1$ will evaluate to the same score, the Buckley algorithm will read the inverted lists for all the terms in the query.

For large collections, the likelihood that two documents will evaluate to the same score is greatly increased, even among documents that are not exactly the same. For example, many legal documents follow specific templates, with only minor changes to the template text. It is likely that many documents generated with the same template will be the same length, and will therefore have the same score for many queries. In order to guard against this effect, the exactness conditions for ranking order must be relaxed. Buckley suggests a method for doing this.

Moffat and Zobel [7] evaluate two heuristics, *Quit* and *Continue*, which reduce the time necessary to evaluate term-at-a-time queries. The *Quit* heuristic dynamically adds accumulators while query processing continues, until the number of accumulators meets some fixed threshold. At this point, documents are ranked by the partial scores in the accumulators and returned to the user. The *Continue* strategy is similar, in that it uses only a fixed number of accumulators. However, when the accumulator threshold is reached, it continues query evaluation, but only considers those documents that already have accumulators allocated. The *Continue* method was found to be particularly effective; at times it was more effective than the baseline system.

Brown [2] presents a method for efficiently finding a small list of candidate documents for scoring. These candidate documents are considered to be the most likely set of documents to appear in the top k results. This short list of candidates can be scored quickly by skipping through inverted lists. To create the candidates list for a query, the search engine takes the union of term-specific candidates lists created at index time. For a given term t, its candidate list contains the top documents in the ranked list for the query t. Brown finds excellent speedups for this approach.

The method presented in this paper uses candidate lists as well, but it uses these lists as a hint for finding top scoring documents. In a query where the terms have extremely low co-occurrence, following only term candidates lists is likely to lead to poor effectiveness. For instance, the best results for the query "howard turtle" are unlikely to come from the candidates for "howard" or "turtle".

Broder, et al. [1], consider query evaluation in a two stage process. First, the query is run as a Boolean *and* query, where a document is only scored if all terms appear. If this process finds at least k documents, the process stops. However, if the number of documents found is less than k, a second query is issued that considers all documents that contain any query term.

Turtle and Flood [8], consider a series of query optimization techniques, including some exact methods and some approximate methods. In the document-at-a-time max_score method, the top candidate document for each term is stored in the index, like the approach taken by Brown, except only one document is stored.

Document-at-a-time max_score works much like a traditional document-at-a-time system, until k documents have been scored. At this point, the smallest score becomes a lower bound for document scores returned for this query. As the query continues, the algorithm maintains this k^{th} smallest score, which will become more accurate as query processing continues. In this paper, we refer to this score as the threshold score.

The max_score algorithm also sorts terms in order by frequency of occurrence. The terms that appear most frequently will contribute the least to document scores. Let the most frequent term be t_0. We hope that at some point during query processing, the threshold score will be large enough to prove that any document appearing in the top k results must contain some query term other than t_0. This allows only documents containing at least one of $t_1, ...t_n$ to be scored. When conditions are favorable, max_score scores only documents that contain a few discriminating terms. This is somewhat like the *Continue* approach of Moffat and Zobel, except the process is dynamic and guarantees correct results.

The term-at-a-time max_score method is similar to the method presented by Buckley. However, Turtle and Flood explore a rank-safe optimization, where terms are evaluated until the top k documents are guaranteed to be in the correct order.

4. APPROACH

4.1 Finding Top Documents

In the query likelihood formulation of language modeling, we rank documents by $P(Q|D)$: the probability that the query was generated by the same language model as a given document. We estimate this probability by considering the probability of generation of each term independently, as follows:

$$P(Q|D) = \prod_{q_i \in Q} P(q_i|D)$$

Since the document is likely to be short, and therefore may not contain enough data to reliably estimate $P(q_i|D)$, we estimate $P(q_i|D)$ by smoothing against the term distribution in the collection, C. For this research, we use linear interpolation as follows:

$$P(q_i|D) = (1-\lambda)\frac{c(q_i;D)}{|D|} + \lambda\frac{c(q_i;C)}{|C|}$$

In this case, $c(q_i;D)/|D|$ represents the number of times the word q_i appears in the document D divided by the length of D. The second term, $\frac{c(q_i;C)}{|C|}$ is similar, but represents the count of q_i in the entire corpus divided by the corpus length.

Note that in the formulation above, only the $\frac{c(q_i;D)}{|D|}$ component varies with respect to the current document; therefore, the documents for which $P(q_i|D)$ is largest will be those for which $\frac{c(q_i;D)}{|D|}$ is largest.

In the term bounded max_score method, we rank all documents in the collection by $\frac{c(t;D)}{|D|}$ for each term t. If we stored the entire ranking, we would have inverted lists sorted by term frequency. However, our goal is for these lists to be quite small, so that they do not substantially increase the size of our index. We call these lists topdocs lists, as they were called in the INQUERY system. In Section 6.1, we test different methods for determining how long these lists should be.

5. ALGORITHM

Our method, term bounded max_score, is related to the max_score work of Turtle and Flood [8]. However, like Brown [2], we use topdocs lists of for each term.

The key to this optimization, like many other methods, is that we are not concerned with ranking every document in the collection; we are only concerned with the top k documents in the ranked list. Our goal is to score as few documents as possible while still finding the top k documents for our query.

At the beginning of query evaluation, we take the union of the topdocs lists for each term. This gives us a candidates list of documents that we must score. We suppose that the lengths of these topdocs lists are short enough that scoring these documents will take a small fraction of the query time.

Based just on the information in these topdocs lists, we compute an approximate score for each candidate document. This score is guaranteed to be less than or equal to the true score for each document. We then take the approximate

Pruning Method	Description
Fixed	The top documents list contains the same fixed number of documents for each term.
Fraction	The top documents list for a term contains some constant fraction of all the documents that contain that term. In this case, very frequent words, like "the", will have more top documents than less frequent words, like "abstruse".
Frequency	The top documents list contains all documents where the term constitutes more than some fixed fraction of the document. For instance, the lists might contain all documents where the term accounted more than 1% of word occurrences.

Figure 2: Pruning strategies attempted in simulation

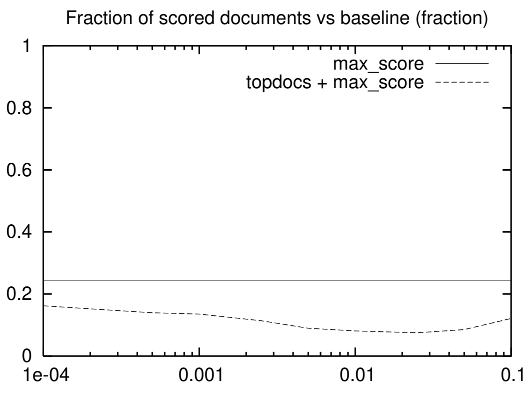

Figure 3: Mean documents scored with max_score and term bounded max_score using the fraction pruning method.

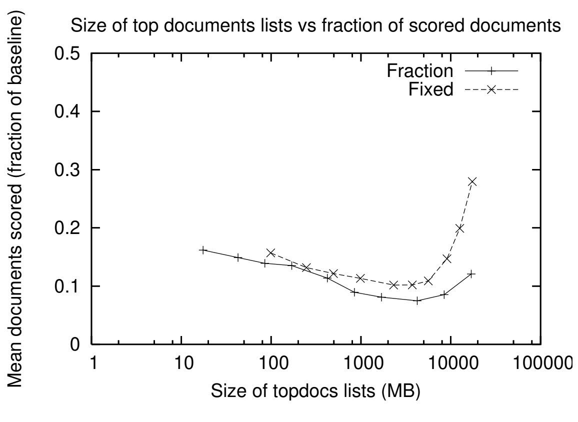

Figure 4: Mean documents scored with the fixed and fraction pruning methods across varying topdocs list sizes.

score of the k^{th} document to be the threshold score; we know that the k^{th} document will have a score at least as high as this one.

We also take the lowest scoring document from each of the top documents lists, and use these documents to bound the score of the remaining documents in the collection. For some document d that contains term t, document d either appears in the top documents list for term t, or it has a lower term score for t than any document in the top documents list for t. This gives us a bound on the term score of any document that is not in the top documents list for t.

We then execute the max_score algorithm, but we use the tighter term score bounds and the threshold computed from the candidates list. We make sure to score every document in the candidates list, plus every document we are directed to score by the max_score algorithm. Like the max_score algorithm, this technique is guaranteed to give exactly the same results as a system that scored every document.

As we show in the experimental section, this optimization gives an excellent increase in performance in practice. One intuition about this effect is that the "good" documents are in the top documents list, and this method pinpoints them more effectively. However, early experimentation does not necessarily support this hypothesis; it is surprising how often a high scoring document is not in any of the topdocs lists.

We prefer to think of the top documents lists as containing the outliers for each term. These high scoring documents keep us from forming a strong bound on the remaining documents in each list. Once we have segregated them from the rest of the data, we get bounds that better reflect the actual data in the collection, and it is these bounds that allow max_score to work more effectively.

We find that these outliers are more prominent in web data than in collections that have been considered in the past. It is not uncommon to find extremely short documents in web collections; sometimes we find documents containing a single word. These single word documents have extremely high term scores, since $\frac{c(w;d)}{|d|} = 1$. A single one-word document containing the word "the" can cause max_score to score hundreds of thousands of documents unnecessarily. The top documents lists remove these outliers from general consideration.

6. EXPERIMENTAL APPROACH

We tested our algorithm on the TREC GOV2 collection, which consists of approximately 25 million web pages crawled from the .gov domain in January 2004. The collection is 426GB uncompressed, and contains 22.7 billion words. We considered using a news corpus as well for comparison, but we did not have access to a news corpus large enough to reliably estimate query timings. We expect that our system is particularly well suited for the irregular data found on the web, and may not perform as well on news corpus data.

We tested our approach using the 50 title queries (701-750) from the 2004 TREC Terabyte track. These queries average 3.02 terms per query, and are content queries, not named-page finding queries. These queries do not contain known stopwords, and every query contains at least two terms.

We indexed the GOV2 collection using Indri, a new scalable language modeling search engine developed at the University of Massachusetts. For our initial experimentation, we exported the inverted lists from this index to an simulation system that recorded the actions of each query algorithm in detail. We used this simulation system to explore different heuristics for finding an optimal length for the top documents lists. We then implemented the term bounded max_score algorithm in Indri, where we verified that this method could achieve real performance gains in a real retrieval system.

6.1 Topdocs List Pruning Methods

We tried three methods for pruning topdocs lists: *fixed*, *frequency* and *fraction* (see Figure 2). In the *fixed* method, we store a fixed number of documents in each topdocs list. In the *frequency* method, we store all documents where $\frac{c(t,d)}{|d|}$ is greater than some constant. In the fraction method, we store some constant fraction of the number of documents that contain a given term.

In all cases, we decided not to use topdocs lists for terms occurring in fewer than 1000 documents, as this provided significant space savings.

6.2 Simulation Results: Title

We evaluated the three topdocs trimming methods in simulation Each method was compared against the baseline, where every document in the collection that contains any query term is scored, and the max_score approach, where only some of these documents are scored.

Each of these pruning strategies has a single parameter. We ran each of the 50 queries using approximately 10 parameter values for each trimming method. We then found the mean number of documents scored for each trimming method parameter setting.

At the best parameter setting, the *frequency* method scored 11.3% of the documents in the collection, while the *fraction* method scored 11.4%, and the *fixed* policy scored 18.1%.

For the *fraction* method, we show its performance across its parameter space (Figure 3) when compared to the baseline and the max_score method. Note that results are normalized to the baseline, so the baseline is the top bar of the graph. The max_score method scores an average of 22% of the collection. The term bounded max_score method with *fraction* pruning scores fewer documents than max_score over its useful parameter space. As the graph shows, the number of documents scored changes only slightly as the fraction of documents in the topdocs lists changes from 0.5% to 5%.

Although the *frequency* approach appears promising, it has the disadvantage that its disk space usage is difficult to predict. The *fraction* method has the advantage that its disk usage is highly predictable–choosing to store 5% of each inverted list in a topdocs list will result in roughly 5% additional disk space used. The *fixed* method is somewhat less

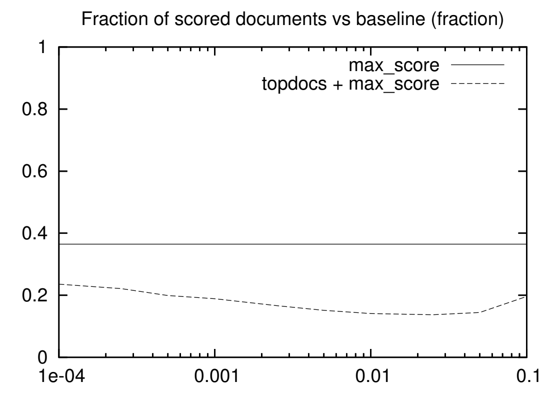

Figure 5: Mean documents scored for expanded queries with the max_score method, and with term bounded max_score with the fraction pruning method.

predictable, since its disk usage is proportional to vocabulary size and not corpus length. However, vocabulary growth has been well studied, and tends to be strongly correlated to corpus length; this makes the *fixed* method predictable as well. Because using only a small amount of disk space was an important goal for us, we decided to only consider the *fixed* and *fraction* methods further.

We compared the performance of the *fraction* and *fixed* methods based on efficiency at different topdocs list lengths. The results are shown in Figure 4. Note that the *fraction* scores fewer documents than the *fixed* method for all topdocs list sizes.

6.3 Simulation Results: Expanded

In order to test our method on longer queries, we took our set of 50 title queries and used query expansion to create a set of 50 10-word queries.

To do this, we ran each title query against the GOV2 collection, and recorded the top 15 documents returned. We sorted the terms in these top 15 documents by order of occurrence, and removed any known stopwords, keeping only the top 10 words. This process generated 50 queries, each with exactly 10 words.

We found that, counter to general wisdom regarding the max_score optimization, max_score and our technique did not work as effectively on our expanded query set as on title queries. In Figure 5, we show results for the *fraction* pruning method and the max_score optimization on the expanded query set. Compared to Figure 3, the graphs have a similar shape, but both max_score and term bounded max_score do not perform as well against the baseline.

To explain why this is so, we looked at the performance individual expanded queries. The following query is one of the poorest performing queries in our query set:

```
loan census transportation application work
area year report state data
```

We have ordered the query terms from least frequent to most frequent in the GOV2 collection; this is the way they would be evaluated by a term-at-a-time scoring system. The following table shows the upper and lower bounds on scor-

Method	Seconds/query	User (s)	User (σ^2)	Elapsed (s)	Elapsed (σ^2)	Documents scored
Baseline	4.339	177.93	0.998	216.92	1.025	112425031
Max_score	2.226	74.15	0.065	111.32	0.025	41697980
Term bounded max_score	1.728	47.77	0.212	86.39	0.157	24300922

Figure 6: Time to evaluate TREC topics 701-750 against the TREC GOV2 collection

Method	Seconds/query	User (s)	User (σ^2)	Elapsed (s)	Elapsed (σ^2)	Documents scored
Baseline	34.70	1473.3	2.12	1735.1	1.72	508223689
Max_score	20.64	779.4	1.03	1031.9	1.14	255740580
Term bounded max_score	15.04	489.1	5.37	752.0	5.55	150479904

Figure 7: Time to evaluate 50 10-word queries against the TREC GOV2 collection

ing contributions of these terms, as found by a traditional max_score run:

term	lower bound	upper bound
loan	-10.2585	-0.2876
census	-10.2190	-0.2876
transportation	-9.5817	-0.2876
application	-8.7923	-0.2875
work	-8.6065	-0.2874
area	-8.1856	-0.2873
report	-8.1148	-0.2873
year	-7.8581	-0.2872
state	-7.1777	-0.2867
data	-6.9300	-0.2864

This query contains many terms that are very close in terms of contribution bounds. In particular, `application`, `work`, `area` and `report` are very similar in terms of their contribution to the final score. This makes it very difficult for us to skip any one of them. In practice, this is exactly what happens; a max_score query without topdocs is only able to skip through three terms (`year`, `state`, and `data`) and must examine every posting of the remaining terms. As `report` and `area` are each in approximately 6 million documents, this leads to a very expensive query.

We expect that, in general, smaller queries are more likely to have a few infrequent terms, paired with perhaps one frequent term. In this case, it is very easy to skip through the frequent term, providing excellent improvement. In expanded queries, it is likely that many of the terms are frequent, which makes it less likely that any one frequent term will be able to be skipped.

It should be noted that many large queries come out of query expansion, where each term in the query is assigned a weight during the expansion process. While these queries can be very long, many of the queries will have weights that are quite small, which will serve to bound the effect that term has on the query score. We suspect that our technique will perform much better on these kinds of queries, and anecdotal evidence seems to support this conclusion.

6.4 Indri Results

In order to verify that our optimization works well in practice, we implemented the term bounded max_score optimization in Indri. For the following tests, we chose the *fraction* topdocs pruning method, taking the top 1% of documents from each inverted list. We chose only inverted lists containing more than 1000 documents. As before, we tested against the GOV2 collection. The inverted lists in our index totaled 44GB, while the topdocs lists took an additional 270MB.

While Indri is capable of running on a cluster of machines, our tests used a single Pentium IV machine, with a CPU clock speed of 2.6GHz. The machine had 2GB of RAM and 1.4TB of storage in a RAID 5 configuration, and was running Red Hat Linux version 9.

For each query method, we ran a warm up run that was not timed, followed by 5 consecutive runs. The times shown are the mean of those 5 runs, in seconds. Each query returned 10 documents, and the time reported includes the time necessary to look up the names of the documents returned from a B-Tree. The variance is shown for each time figure reported.

The results for the GOV2 title queries are reported in Figure 6. We see a 23% improvement over max_score and 61% over the baseline in time per query on this task. In Figure 7, the improvement is 28% over max_score, 57% over the baseline. In the title query case, the baseline scores 4.62 times as many documents as our method, while in the long query case, this factor drops to 3.39. We believe that certain constant factors, like the time to look up document names and the time to decompress blocks of inverted list postings, account for the reason that user CPU time does not track documents scored closely.

Our time of 1.7 seconds per title query is competitive with the fastest standard term-at-a-time systems at this year's TREC Terabyte task. The fastest single machine entry, from SABIR Research, averaged 1 second per title query. The second fastest system, from RMIT, using the Zettair retrieval system, took 2 seconds per title query. All query times at TREC were rounded to the nearest second, so it is difficult to make precise comparisons here. Given the data that we have, our system appears to be competitive.

7. CONCLUSION

In this work, we have shown that significant query optimization is possible in document-at-a-time systems, even beyond the max_score method. We achieve a 23% speedup versus max_score and 61% versus scoring all documents that contain any of the query terms. We find that this method makes it possible to get good query performance with document-at-a-time systems. Our method, term bounded max_score, allows our Indri system to quickly evaluate simple queries while still handling more complex query operators.

8. ACKNOWLEDGEMENTS

We are grateful for comments from the anonymous reviewers, as well as Donald Metzler, Kevyn Collins-Thompson and Jamie Callan, whose comments were invaluable for strengthening this paper.

This work was supported in part by the Center for Intelligent Information Retrieval and in part by Advanced Research and Development Activity and NSF grant #CCF-0205575 . Any opinions, findings and conclusions or recommendations expressed in this material are the author(s) and do not necessarily reflect those of the sponsor.

9. REFERENCES

[1] A. Z. Broder, D. Carmel, M. Herscovici, A. Soffer, and J. Zien. Efficient query evaluation using a two-level retrieval process. In *Proceedings of the twelfth international conference on Information and knowledge management*, pages 426–434. ACM Press, 2003.

[2] E. W. Brown. Fast evaluation of structured queries for information retrieval. In *Proceedings of the 18th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 30–38. ACM Press, 1995.

[3] C. Buckley and A. F. Lewit. Optimization of inverted vector searches. In *Proceedings of the 8th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 97–110. ACM Press, 1985.

[4] J. P. Callan, W. B. Croft, and S. M. Harding. The INQUERY retrieval system. In *Proceedings of DEXA-92, 3rd International Conference on Database and Expert Systems Applications*, pages 78–83, 1992.

[5] D. E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming*. Addison-Wesley, Reading, Massachusetts, 2nd edition, 1998.

[6] D. Metzler and W. B. Croft. Combining the language model and inference network approaches to retrieval. *Information Processing and Management*, 40(5):735–750, September 2004.

[7] A. Moffat and J. Zobel. Self-indexing inverted files for fast text retrieval. *ACM Trans. Inf. Syst.*, 14(4):349–379, 1996.

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

[9] I. H. Witten, T. C. Bell, and A. Moffat. *Managing Gigabytes: Compressing and Indexing Documents and Images*. John Wiley & Sons, Inc., 1994.