

Efficient Document Retrieval in Main Memory

Trevor Strohman
strohman@cs.umass.edu

W. Bruce Croft
croft@cs.umass.edu

Department of Computer Science
University of Massachusetts
Amherst, MA 01003

ABSTRACT

Disk access performance is a major bottleneck in traditional information retrieval systems. Compared to system memory, disk bandwidth is poor, and seek times are worse.

We circumvent this problem by considering query evaluation strategies in main memory. We show how new accumulator trimming techniques combined with inverted list skipping can produce extremely high performance retrieval systems without resorting to methods that may harm effectiveness.

We evaluate our techniques using Galago, a new retrieval system designed for efficient query processing. Our system achieves a 69% improvement in query throughput over previous methods.

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

General Terms: Algorithms, Design, Experimentation, Performance

Keywords: Impact-sorted indexes, Memory

1. INTRODUCTION

The commodity computer hardware industry has seen two major changes in the past four years. First, 64-bit x86 processors have entered the market, breaking the 4GB address space barrier for commodity systems. Second, increases in processor clock speed have come to an abrupt halt, replaced by a industry-wide push to put many processor cores on a single die.

The first change gives us enough virtual address space to store the entire inverted file of even large collections. Storing inverted files in memory was potentially possible before on certain 32-bit processors, but required complicated programming. The 64-bit address space of modern processors makes this simple. The second change means that, in order to achieve top performance on modern processors, we must use many different threads simultaneously. If each thread

represents a different query, it is not clear that disks will be able to keep up.

Optimizing disk-based information retrieval systems requires a delicate balance between reading less data and requiring fewer random jumps through the data. Reading less data is an obvious goal, since reading fewer bytes means fewer bytes to process, but random disk seeks are so expensive that skipping to skip portions of data must be done delicately.

We instead consider storing the entire index in memory. Storing the index in memory greatly minimizes the cost of random reads, allowing us to focus entirely on reading fewer bytes.

We provide the following research contributions in this paper:

- We present a new method for continuous accumulator pruning in impact-sorted indexes. Our method increases query throughput by 15% over the method proposed by Anh and Moffat [3] while still remaining *rank safe* results (i.e. documents are returned in the same order that they would be in an unoptimized evaluation)
- We show how our accumulator pruning technique can be combined with inverted list skipping to achieve a 69% total increase in throughput while maintaining the *rank safe* property.
- We provide a technique for optimizing the appropriate skipping distance to use during index based on simulation. Unlike all previous work we are aware of, we consider different skipping distances for different list lengths. We show that using list-length-dependent skip lengths can improve query throughput slightly.
- We show that storing inverted lists in memory can significantly improve performance, adding to previous results from Büttcher and Clarke [7]. We show that the algorithm presented by Anh and Moffat [3] can evaluate queries 7 times faster on our system than the speed quoted in their paper.
- We provide additional insight into implementation techniques necessary for efficient performance for in-memory retrieval systems. In particular, our results indicate that in-memory processing may be more economical than disk-based systems for some tasks.

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.

SIGIR '07, July 23–27, 2007, Amsterdam, The Netherlands.

Copyright 2007 ACM 978-1-59593-597-7/07/0007 ...\$5.00.

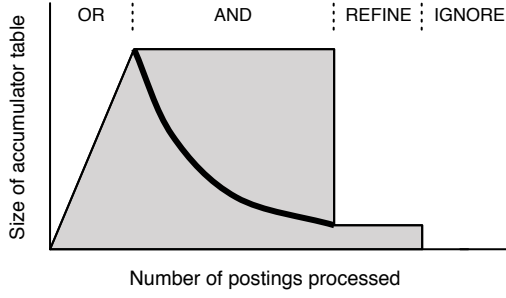


Figure 1: Relative number of accumulators used during the query evaluation process. The gray filled area represents the usage pattern in Anh and Moffat. The thick solid line represents the decreased accumulator usage of our approach.

2. ALGORITHM

Over the past six years, impact-sorted indexes have been shown to be an effective and efficient data structure for processing text queries [1, 2]. These indexes store term weights directly in the index, like the SMART system [6]. However, impact-sorted indexes use a very small number of distinct term weights; in this paper we use just 8 different values. The small number of values used allows these indexes to store documents in impact order while still allowing for very high level of compression [1].

To generate effective retrieval results, care must be taken in selecting the impact values assigned to each term. Many different approaches are possible for this task. Anh and Moffat have suggested a method for truncating BM25 values into integers, and later introduced a document-centric model.

We use the Anh and Moffat document-centric impact model of for query evaluation [2]. The details of this model can be found in the references. For the purposes of this paper, the important aspects of this model are that each term in a document receives an integer weight, based primarily on document statistics. This is roughly analogous to TF in a TF-IDF formulation. At query time, a query weight is computed which is analogous to IDF. We call the document weight $w_{t,d}$ and the query weight $w_{t,q}$.

The retrieval score S_d for a document d , evaluated with a query Q is defined as:

$$S_d = \sum_{q \in Q} w_{t,q} w_{t,d} \quad (1)$$

The index contains an inverted list for each term t . In impact-sorted indexes, the list is separated into segments, one for each distinct value of $w_{t,d}$. Each segment contains a list of documents that share the same $w_{t,d}$ value.

The accumulator tuple we consider for this work is the same as used by Anh and Moffat. Each accumulator is a triple $\langle T, s, d \rangle$, composed of a term set T , a score s , and a document number d . The document number d and score s are traditional: the score s records the partial score for the document d based on query processing so far. The term set T records all terms that have been scored for this document so far.

In the code shown in this section, an accumulator for a particular document d is denoted A_d . This symbol can be

used as a score or as a set of terms, but the usage should be clear based on context.

Figure 1 describes the different phases of the Anh and Moffat algorithm we refer to in this section, as well as our new algorithm. The reduced accumulator usage of our algorithm is shown by the dark black line.

All of the query evaluation methods we consider in this paper follow the pseudocode shown below, although with different implementations of ProcessSegment, TrimAccumulatorList, and CanQuit.

```

procedure PROCESSQUERY( $Q$ )
   $A \leftarrow \{\}$  ▷ The accumulator table
   $S \leftarrow \{\}$  ▷ List of accumulator segments
  for all query terms  $t \in Q$  do
     $I_t \leftarrow$  inverted list for term  $t$ 
     $w_{t,q} \leftarrow$  weight for term  $t$ 
    for all segments  $I_{t,s} \in I_t$  do
      add  $\langle t, w_{t,q} \times w_{t,d}, I_{t,s} \rangle$  to  $S$ 
    end for
  end for
  sort  $S$  in descending order by segment score
  for all segments  $\langle t, w, I_{t,s} \rangle$  in  $S$  do
    ProcessSegment( $A, w, I_{t,s}$ )
    TrimAccumulatorList( $A, S$ )
    if CanQuit( $A, S$ ) then
      break
    end if
  end for
  sort  $A$  by score, return top  $k$  results
end procedure

```

The simplest form of evaluation evaluates every posting in every segment. We define ProcessSegment below. TrimAccumulatorList is set to an empty function, and CanQuit is set to a function that returns false on all inputs. In unoptimized evaluation, the system is always in OR mode.

```

procedure PROCESSSEGMENT( $A, w, I_{q,s}$ )
  for all documents  $d$  in segment  $I_{q,s}$  do
    if  $A_d \notin A$  and OrMode = true then
       $A_d \leftarrow 0$ 
    end if
    if  $A_d \in A$  then
       $A_d \leftarrow A_d + w$ 
    end if
  end for
end procedure

```

2.1 AND Processing

If we know that all documents that could possibly be in the top k are already in A , we can update only accumulators that already exist. This is called AND mode, as opposed to the unoptimized OR mode. When in AND mode, ProcessSegment never adds new accumulators to the accumulator table. This is beneficial both because we do not incur the costs of building new accumulators, and because A stays small, leaving fewer accumulators to update and sort later.

The Anh and Moffat algorithm automatically detects when AND processing can be used safely by monitoring two quantities: a threshold value τ and a remainder function ρ . We describe these next.

The threshold value τ is a lower bound on the score of the last document that will be displayed to the user. Here, we assume that k represents the number of documents re-

quested. We can compute a reasonable τ value by using the k^{th} largest score in the accumulator table A . Since the score in any given accumulator is monotonically increasing, τ represents a lower bound on the score of the k^{th} retrieved document at the end of an unoptimized retrieval. If k accumulators do not exist yet, $\tau = 0$.

For example, suppose at some point during the retrieval process, the k^{th} largest score found in any accumulator was 50. Therefore, $\tau = 50$. Since none of the scores in the accumulator table can go down, we know that every document in the k returned to the user will have a score of at least 50.

The second monitored quantity is the score remainder function, ρ . This function computes an upper bound on the total additional amount of score that an accumulator could possibly gain through further processing of the inverted lists.

As an example, suppose we know the accumulator for some document d currently contains a score of 15, and that we are processing a four term query: $t_1 t_2 t_3 t_4$. The accumulator also records that we have already seen postings for document d in the inverted lists for t_1 and t_3 . Any remaining score for document d must come from the inverted lists for t_2 and t_4 . If we know that all remaining postings for t_2 have scores less than 5, and all remaining postings for t_4 have scores less than 6, we know that this accumulator can only increase by 11. Therefore, $\rho(\{t_2, t_4\}) = 11$.

We define the function ρ as follows:

- $\rho(\{t_i\})$ = the largest score remaining in the inverted list for term t_i
- $\rho(T) = \sum_{t \in T} \rho(\{t\})$

Remember that Q is the set all terms in the query. When $\tau > \rho(Q)$, we know that no more accumulators need to be created. This is true because no new accumulator will ever achieve a score greater than $\rho(Q)$, but a score of τ is necessary to enter the ranked list shown to the user.

With τ and ρ defined, we can now define a pruned processing algorithm:

```
procedure PROCESSSEGMENTPRUNED( $A, q, w, I_{q,s}$ )
  OrMode  $\leftarrow \tau < \rho(Q)$ 
  ProcessSegment( $A, q, w, I_{q,s}$ )
end procedure
```

2.2 Trimming Accumulators

Computing τ and ρ allows us to stop adding accumulators to A , but they can also help us identify accumulators that can be safely removed from the table.

As query evaluation continues, τ grows and ρ falls. If there comes a time when an accumulator A_d contains a score low enough that no additional postings could cause its value to rise above τ , we know it can never enter the final ranked list. Therefore, we can prune it from consideration.

```
procedure TRIMACCUMULATORLIST( $A, S$ )
  for all accumulators  $A_d$  in  $A$  do
    if  $A_d + \rho(A_d) < \tau$  then
      remove  $A_d$  from  $A$ 
    end if
  end for
end procedure
```

The Anh and Moffat algorithm trims the accumulator table just once, when the top k results have been determined. Instead, our algorithm trims accumulators after each inverted list segment is processed. This difference is highlighted in Figure 1 by the dark black line.

Having a smaller accumulator list improves speed because there are fewer accumulators to update. However, when the inverted list segments become much longer than the table of accumulators, inverted list skipping becomes possible. Since the inverted list is stored in document order, if we also store the accumulator table in document order we can identify when large sections of the inverted list are not worth decoding. By using skipping information, discussed in more detail later, we can skip over those regions without decompressing them.

2.3 Ignoring Postings

At some later point during query processing, it may be possible to determine the final order of the top k results without continuing to process postings. For this to happen, two conditions must be satisfied:

- The top k documents must be fixed (that is, the identity of the top k documents is not in question)
- No additional postings may be able to change the ordering of the top k documents.

We can check the first condition by checking all existing accumulators. All accumulators A_d containing scores less than τ must satisfy the condition $A_d + \rho(A_d) < \tau$. Essentially this means that if the accumulator is not in the top k now, it never will be, no matter how many additional postings we process. Note that it is not sufficient to check that $|A| = k$, since many documents may share the same score.

We check the second condition by reviewing all accumulators with values of at least τ . If the accumulator of the document currently in rank i could surpass the document in rank $i - 1$ after processing additional postings, we cannot stop processing.

These checks lead to the following algorithm, which is used in both our algorithm and Anh and Moffat:

```
procedure CANQUIT( $A, S$ )
  for all accumulators  $A_d$  in  $A$  do
    if  $A_d < \tau$  and  $A_d + \rho(A_d) \geq \tau$  then return False
  end if
end for
   $A' \leftarrow$  all accumulators  $A_i$  in  $A$  such that  $A_i + \rho(A_i) > \tau$ 
  sort  $A'$  in ascending order by score
  for all accumulators  $A'_i$  in  $A'$  do
    if  $A'_i = A'_{i+1}$  and  $\rho(A'_i) > 0$  then return False
    else if  $\rho(A'_i) > A'_{i+1} - A'_i$  then return False
  end if
  end for
end procedure
```

3. IMPLEMENTATION

We performed the experiments in this paper using the Galago¹ retrieval system. We created the Galago system to help investigate indexing techniques that enable efficient query processing. Galago consists of a flexible distributed indexing system, written in Java, and an optimized query processing component written in C++.

¹<http://www.galagosearch.org>

3.1 Indexing

Our system follows the impact-ordered index design proposed by Anh and Moffat [1]. Inverted lists are stored in order of impact value. The impact value is encoded first, followed by a length value. After that, the document numbers that share the same impact value are delta encoded. As a result, each inverted list is separated into logical blocks, one for each bin value. Each block is processed atomically during query evaluation. This method produces indexes that are size-competitive with other space-efficient methods, typically around 7GB for the GOV2 collection.

The inverted file is compressed using standard variable byte encoding [18]. The vocabulary is compressed using 15-of-16 encoding, plus additional prefix encoding. The inverted file is segmented into blocks of approximately 32K in size, although no term spans multiple blocks. Any term with more than 32K of inverted list data gets its own block. An abbreviated vocabulary table is used to look up the appropriate block for a given term. A block contains a sub-vocabulary that can be efficiently searched to find the appropriate inverted list. This blocking technique efficiently packs infrequent terms together, so that the abbreviated vocabulary table can be very small. In our experiments, the vocabulary table required just 2MB of space. A similar technique has been used previously by Büettcher and Clarke [7].

Our indexer was built for simplicity, experimentation and extreme parallelism. We used an approach that combines ideas from early information retrieval systems with principles from Dean and Ghemawat's MapReduce [11] in order to make a simple, parallel indexer. The first stage processes text documents, converting them into compressed lists of (document, word, count) tuples. The next phase combines these tuples in order to determine the inverse document frequency for each term. In parallel, another process combines the list of document names into a single table. The binning stage follows, where the IDF table is combined with the parsed word count tuples to generate binned term weights (in this case, document centric impacts). The final stage merges the binned lists together into a single index.

Indexing the compressed GOV2 collection requires 60 hours of CPU time, with 42 hours devoted to parsing, 16 for binning (impact generation), and the remaining 2 hours for merging. However, the parsing and binning stages are massively parallel. In practice, we can build a GOV2 index in 4 hours using roughly 20 processors (this quantity varies, as the grid of processors is shared for other research tasks). The skipping information in the inverted lists is written in the final merge stage, so the skipping parameter can be changed with only two hours of additional work once an index has been built.

The relatively long indexing time required by our system should not be a reflection of the optimized indexing time for this task. Previous work in this area indicate that impact-sorted GOV2 indexes can be built in under 7 hours on a typical desktop computer using optimized implementations [3].

3.2 Retrieval

We take advantage of the 64-bit address space of new commodity machines to memory map the inverted list file into the virtual address space of the process. This allows multiple retrieval processes to run simultaneously while sharing the same memory pages.

For these experiments, each accumulator is a 64-bit value. The term set T is encoded as an 8-bit bitmap. The document score consumes 24 bits of the space, leaving the remaining 32 bits for the document identifier. For a query with n terms, where $n > 8$, the system would process the shortest $n - 8$ inverted lists completely, without optimization, leaving the 8 bitmap bits for the longer lists. More recent versions of Galago switch to larger accumulators for longer queries.

The accumulators are stored in an array, sorted by document identifier. Packing the accumulators closely gives us good cache performance, and the document ordering makes the skipping optimization possible. This accumulator organization forces Galago to copy the accumulator table once for each inverted list segment processed; however, like Bast et al., we found the sorted array to be preferable to hash tables for this task [4].

4. CHOOSING SKIP LENGTHS

When adding skip information to an index, it makes sense to ask how long the skip distance should be. Moffat and Zobel [14] suggest a method for determining this parameter. We use a slightly different formulation in this paper which remains in the same spirit. We suppose that an inverted list segment is b bytes long and k entries exist in the accumulator table. We also suppose that there are b_1 skip pointers, each of which can be encoded in 4 bytes, and that each skip pointer skips d_b bytes. Therefore, the expected total number of bytes processed is:

$$4b_1 + \frac{kd_b}{2}$$

The second term estimates the number of bytes decompressed in the inverted list. In reality, we will never decompress a byte of the inverted list more than once, so this has a natural upper bound of b :

$$4b_1 + \min\left(\frac{kd_b}{2}, b\right)$$

If the number of accumulators k is longer than the length of the inverted list segment, the system can quickly acknowledge this situation and ignore the skip information. This gives us the final revised time estimate:

$$T(b, k, d_b) = \begin{cases} b & \text{if } k > b \\ 4b_1 + b & \text{if } k \leq b \text{ and } \frac{kd_b}{2} > b \\ 4b_1 + \frac{kd_b}{2} & \text{otherwise} \end{cases} \quad (2)$$

We estimated d_b using data collected from processing the TREC 2005 Efficiency Track queries. For each inverted list segment the system processed while in AND mode, the system recorded the length of the segment b and number of accumulators k that the system had stored in the accumulator table at that time. This provided us with over 500,000 data points to use in simulation.

Note that b and k have an important relationship; if b is large, the inverted list is probably also large. If so, it is likely to be processed at the very end of the query, when the accumulator table is almost empty. If b is smaller, we expect larger values of k . This suggests that different values of d_b should be used for different inverted list segment lengths.

Figure 2 shows this effect. For small segment lengths (about 1000 bytes long), skipping data helps very little, even at very small skip lengths. For moderate lengths (10^4 to

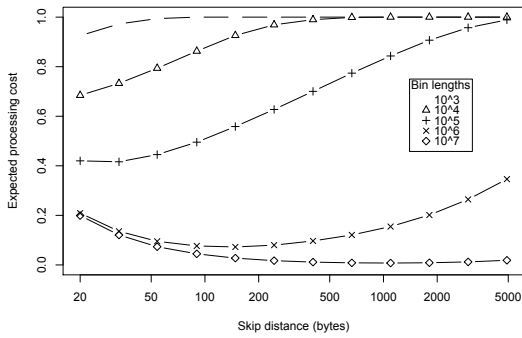


Figure 2: The expected costs, relative to no skipping, of processing segments of varying lengths with varying skip sizes, based on TREC 2005 Efficiency Track queries.

10^5 bytes long), very short skip distances give the largest expected performance increase. For long lengths (over 10^6 bytes long), short skip distances help, but longer skip distances are even better.

The relative frequency of encountering list segments of these different lengths is not shown in the graph. Analysis of our logs indicate that 95% of all inverted list bytes processed in AND mode are in segments larger than 100K. This leads us to consider skip lengths between 50 and 200 bytes.

The analysis in this section rests on the approximation that all byte accesses cost a similar amount. Of course, this is not the case. Reading a single byte from an address that is not currently in cache is expensive, but the process of fetching that byte will cause L1 and L2 cache lines to fill. After a single miss, nearby byte accesses are inexpensive.

Previous work [18] indicates that skip information should be interleaved with inverted list data, but note that this leads to inefficient cache usage; reading a single skip entry of 3 bytes results in reading an additional 29 bytes of unwanted inverted list data. By storing skip information densely and separately from inverted list data we avoid this problem.

5. EVALUATION

We used the TREC GOV2 collection along with the TREC Terabyte Ad Hoc and Efficiency topics for evaluation. The GOV2 collection consists of 25.2 million web pages crawled from the .gov Internet domain. The text data of these web pages consumes 426GB of space.

We processed the GOV2 collection using the Porter2 English stemmer,² and used a common list of 600 stopwords.³

In order to ensure that our system was producing reasonable results, we used the TREC Terabyte Track ad hoc queries to evaluate its effectiveness. These results are shown in Table 1. As is typical for mean average precision evaluation, the system returned 1000 results for each query. We computed mean average precision and precision at 20 figures for each set of ad hoc queries. The results here are close to, although slightly below, efficient systems participating in TREC. We suspect that these results will improve

²<http://snowball.tartarus.org/algorithms/english/stemmer.html>

³<http://goanna.cs.rmit.edu.au/~jz/resources/stopping.zip>

Query set	MAP	P@20
Topics 701-750 (2004)	0.2460	0.4949
Topics 751-800 (2005)	0.3004	0.5290
Topics 801-850 (2006)	0.2592	0.4590

Table 1: Effectiveness on TREC Ad Hoc Queries

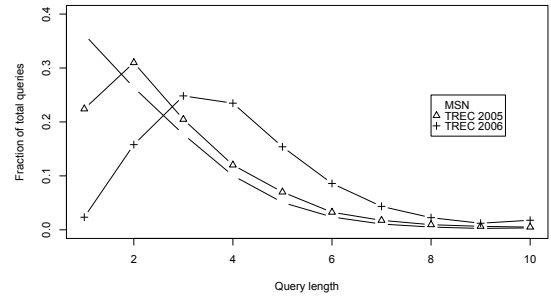


Figure 3: Distribution of query lengths (before removing stopwords) across collections.

as we improve our document parser. Since all optimizations considered in this paper are rank safe, these numbers represent the effectiveness of the system in all optimization modes, and any improvement in baseline effectiveness would be reflected in the optimized modes.

We used C++ preprocessor statements and conditional compilation to build a separate executable for each query optimization method we tested. Using conditional compilation allows the compiler freedom to produce the most efficient code for each query optimization technique.

We used the same computer for all retrieval experiments. Our test machine is worth approximately US\$3000. It contains two dual-core Intel Xeon 5050 “Dempsey” processors, for a total of four CPU cores. Each core runs at 3GHz, and has a dedicated 2MB L2 cache. The cores share 8GB of RAM over a 667MHz bus. All experiments were done in 64-bit mode, in order to permit the inverted file to fit in the virtual address space.

Before timed runs, we ran a simple tool that memory mapped the inverted file and read it from start to finish, in order to ensure the inverted file was completely loaded into memory.

The results of our optimization experiments are shown in Tables 2 and 3. The system returned the top 20 document results for each query. The mean query time for each method (except the unoptimized method) was measured by running each query set three times in immediate succession, then computing the mean query time for the batch. In all cases, total variation between runs of the same method was less than 2%. Because of the wide gulf in processing time between unoptimized and optimized methods, the unoptimized version was run just once.

We measured execution time for individual queries based on log messages produced by the query evaluation code. These messages add approximately 10% to total runtime, and measure many aspects of query processing. Each log message contains a timestamp, which we used to measure the speed of each individual query; the mean times shown

Method	All queries		Query length (terms)									
	Mean	Throughput	1	2	3	4	5	6	7	8	9	10
Query count	47,543	47,543	10,899	17,347	10,888	5,489	1,965	683	233	32	6	1
Unoptimized	317.2	3.2	22	144	410	724	1149	1535	1972	3499	3197	1181
Anh/Moffat	19.0	57.8	0.2	5.5	21	44	79	119	178	366	376	89
Trimming	15.0	66.7	0.1	5.6	18	37	67	105	166	397	362	44
Trimming+Skips	10.3	97.5	0.2	2.9	10	24	49	84	145	360	342	35

Table 2: TREC 2005 Efficiency Queries, average query execution times, in milliseconds. Throughput is measured in queries per second.

Method	All queries		Query length (terms)									
	Mean	Throughput	1	2	3	4	5	6	7	8	9	10+
Query count	99,650	99,650	3,926	23,430	33,122	23,488	10,283	3,515	1,151	407	167	161
Unoptimized	1006	1.0	50.4	212.8	679	1364	2128	2855	3788	4751	5000	9644
Anh/Moffat	60.0	16.7	0.4	5.7	29	73	137	218	319	440	558	3014
Trimming	49.1	20.4	0.4	6.2	24	58	110	180	268	395	586	3180
Trimming+Skips	39.0	25.7	0.4	3.5	14	40	86	150	232	355	544	3122

Table 3: TREC 2006 Efficiency Queries, average query execution times, in milliseconds. Throughput is measured in queries per second.

Skip Length	Throughput		
	Mean	Low	High
64	98.94	98.80	99.20
96	94.73	93.99	95.45
128	97.95	97.60	98.18
256	97.05	96.68	97.36
Model	99.21	98.44	99.83
Model (Large)	99.14	99.05	99.27
Model (Small)	98.69	98.40	98.96

Table 5: Efficiency at varying skip lengths, TREC 2005 Efficiency Queries

here are a result of aggregating that data. Because of this overhead, the timings for various query lengths should be considered somewhat less reliable than the overall average speed figures, which were computed from experiments that did not use logging.

In the TREC 2005 log, we found 2457 queries that did not match any documents in the collection. In the TREC 2006 log, we found 350 queries like this. Although the full query log was processed in each test run, we averaged query performance only over the count of queries with at least some results returned.

In all of the experiments except those about skipping lengths, the index was built with a fixed skip length of 128 bytes.

To measure performance on multiple processor cores, we ran multiple processes of the retrieval system simultaneously on the TREC 2005 query set. To avoid possible interaction between processes, each core processed the queries in random order. We measured the elapsed time between starting the first query process until the slowest query process completed. The number of queries executed was divided by total elapsed time to produce a throughput number. We did not compute a mean query time in this case, since such a number would be misleading: adding multiple processors does not decrease individual query latency, but it does improve overall throughput.

5.1 Analysis

Figure 3 compares the distribution of query lengths in three different query logs. Note that the lengths reported here are measured as the number of whitespace breaks in the query plus 1; it therefore counts stopwords and words that do not appear in the collection. The query lengths reported in other tables refer to the number of inverted lists successfully fetched from the index. The TREC 2005 and TREC 2006 Efficiency queries are used for performance analysis, while the MSN query log data is provided for comparison purposes.⁴ The TREC 2005 queries are a reasonable match for the actual distribution of query lengths on the web.

The gulf between all of the optimizations tested here and a full evaluation is striking, especially in the case of particularly short queries. Notice how all optimizations here complete single term queries in under a millisecond. With no disk seek overhead, the system can jump immediately to the necessary inverted list. After twenty results are read, query processing halts. The unoptimized system is forced to read the entire inverted list and create an accumulator for every document in it, which takes much more time. The percentage difference in query time falls as the queries grow longer, but the Anh and Moffat approach completes queries in half the time of the unoptimized case in all cases, while our method completes them in less than a quarter of the unoptimized time.

Our experiments with different skip lengths show surprisingly similar performance. We ran each query set three times in succession, with the mean throughput shown, as well as the lowest and highest throughput recorded. The differences between many of these settings are well within the range of variability of our tests. The small differences here are reasonable given the results in Figure 2; note that the performance predicted for the largest inverted list segments (the most costly ones) are remarkably flat across differing skip lengths. However, we note that using the results suggested by our model (varying skip lengths based on inverted list seg-

⁴http://research.microsoft.com/ur/us/fundingops/RFPs/Search.2006_RFP_Awards.aspx

Method	1 core		2 cores		4 cores	
	Throughput	Speedup	Throughput	Speedup	Throughput	Speedup
Anh/Moffat	57.8	1.00	108.3	1.87	181.6	3.14
Trimming	66.7	1.00	121.1	1.81	178.2	2.67
Trimming+Skips	97.5	1.00	161.2	1.65	228.8	2.35

Table 4: Speedup when using multiple cores. Throughput is measured in queries per second, while speedup is measured relative to each algorithm’s performance on a single processor.

ment length) results in a slight increase in performance. The large model came from manually slightly increasing the skip lengths suggested by the model, while the small model came from manually slightly reducing the skip lengths suggested by the model. These results suggest that the predicted best setting is close to optimal, but variability in measurement does not allow us to say this with confidence.

5.1.1 Multiple Cores

Our multiple process experiments show the limitations of shared memory bandwidth. We used a machine that is known to be bandwidth starved, and this shows in this experiment. Using four processors simultaneously, the Anh and Moffat algorithm is able to process 3.14 times as many queries as when just one processor is used. This speedup is much higher than the 2.35 speedup of our best algorithm. When four cores are used, the Anh and Moffat algorithm equals the performance of the Trimming version of our algorithm.

We leave a detailed explanation of these numbers for future work, but it is clear that linear scalability is not assured with multiple processors, even when no disks are involved. However, notice that the Anh and Moffat algorithm and the Trimming algorithm access approximately the same amount of memory during evaluation, while the Trimming+Skips algorithm accesses less memory. We believe that the reduced memory access allows Trimming+Skips to maintain its edge over the other algorithms, even when memory bandwidth is scarce.

6. RELATED WORK

Impact-sorted indexes are described in a series of papers by Anh and Moffat [1, 2, 3]. These indexes are a natural next step following the frequency-sorted indexes of Persin et al. [15]. The frequency-sorted indexes suggested by Persin et al. stored term counts instead of discretized document scores, and so these indexes still required query-time length normalization. Additionally, this meant that inverted list entries were not necessarily in score order. However, sorting lists by frequency allowed for a compact and compressible index representation that was amenable to early termination, and formed a basis for the impact-sorted work.

Roughly at the same time as the first impact-sorted work, Fagin et al. proposed a class of algorithms known as threshold algorithms [12]. These algorithms, like the ones shown in this paper, provide a method for efficiently computing aggregate functions over multiple sorted lists by maintaining statistics about the data that remains to be read. This work also considers the added possibility of random access to elements in a list, which is somewhat similar to the skipping process we propose here. From an information retrieval perspective, this work can be seen as a combination of the max score work of Turtle and Flood [17] combined with the

frequency and impact sorted work of Persin et al. and Anh and Moffat [1, 15]. Both Brown and Strohan et al. considered supplemental lists of top scoring documents during query evaluation which can also be considered part of this tradition. [5, 16].

It is possible to have some of the benefits of impact-sorted lists while still sorting by document. Lester et al. [13] show how the memory footprint of accumulators can be significantly reduced without loss of effectiveness. Their algorithm scans inverted lists in document order, but processes only postings with term counts larger than some threshold. As in this work, smaller accumulator sets lead to faster query processing.

In our work, we process less index data by organizing the index for easy skipping and query termination. Another way to process less data is to store less data in the index. Static pruning methods remove information from the index that is unlikely to affect query effectiveness. Carmel et al. considered this process [9]. More recently, Büttcher and Clarke considered index pruning specifically so that the resulting index would fit in memory, although supplemental disk indexes are sometimes used for additional information. Query performance improves in part because of memory speed and in part because of the smaller amount of data, although these different factors were not analyzed in detail. More recent results from the TREC 2006 Terabyte Track shows that other researchers have considered static pruning [8].

While the actual process of storing precomputed scores in lists is not the subject of this paper, there are many examples in the literature of researchers doing this. Cleverdon remarks how, in early experiments, human indexers were asked to choose integer term weights for documents for later use in automatic retrieval [10]. The SMART retrieval system, at least in the 1980s and beyond, used floating point weights stored directly in inverted lists for fast and flexible document scoring possibilities at query time [6]. More recently, Anh and Moffat have proposed two separate schemes for generating term weights; one involving assigning ranges of BM25 scores to integer values, and another using a document-centric approach. [1, 2]

Bast et al. extend the threshold algorithm ideas of Fagin et al. with an enhanced disk IO cost model [4]. Their system contains a score-ordered index as well as a document-ordered index (or, more accurately, an index of values accessible by document identifier). The system processes information in score order until the cost model indicates that it is more efficient to refine remaining accumulators by random access to score information using the document-keyed index. By contrast, our algorithm always accesses data in score order, but additional skip information allows us to jump rapidly to required information when the number of active accumulators drops, without requiring a separate index copy. Like us, the authors use a machine with 8GB of RAM to test their

system, but data duplication causes their indexes to be too large to fit in system RAM.

While memory-optimized processing is a relatively new field for information retrieval research, it is well studied in the database community. The MonetDB system is the traditional example of a main-memory database system [19]. MonetDB stores relational data by column instead of by row, which significantly increases the speed of certain classes of data warehousing database transactions. Research on the MonetDB system has shown that there are important data processing advantages to working in a RAM-only system.

7. CONCLUSION

We have presented a study of efficient query processing techniques in main memory. Our best technique improves query throughput by 69% over a strong baseline. In the process of developing this method, we have shown how log data can be used to determine optimal skip lengths in inverted lists based on an estimation of total bytes read.

The results shown here convince us that memory-based systems are not only more efficient, but more economically sound than disk-based systems for efficient retrieval. While systems capable of processing the GOV2 data are available for less than US\$1000, these systems are typically not capable of processing more than 10 queries per second without resorting to unsafe optimizations (early termination and/or static pruning). On a system that costs less than 4 times as much, our algorithm produces throughput rates almost 10 times as fast on a single processor, and 20 times as fast when multiple processor cores are used. However, our multiple core experiments show that more processors do not guarantee scalability: high memory bandwidth is critical for high performance.

The combination of fast random access in memory and skip information in inverted lists allows us to achieve performance that approaches heavily statically pruned systems, but without the potential loss in effectiveness [7]. Essentially the skipping information allows us to prune the index dynamically at query time, resulting in efficient retrieval performance.

8. ACKNOWLEDGMENTS

Thanks to Donald Metlzer, Holger Bast, Stefan Büttcher, and the anonymous reviewers for their helpful comments on this paper. Thanks also to Vo Ngoc Anh and Alistair Moffat for clarifications on their work.

This work was supported in part by the Center for Intelligent Information Retrieval, in part by NSF grant #CNS-0454018, and in part by ARDA and NSF grant #CCF-0205575. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect those of the sponsor.

9. REFERENCES

- [1] V. N. Anh, O. de Kretser, and A. Moffat. Vector-space ranking with effective early termination. In *SIGIR 2001*, pages 35–42, New York, NY, USA, 2001. ACM Press.
- [2] V. N. Anh and A. Moffat. Simplified similarity scoring using term ranks. In *SIGIR 2005*, pages 226–233, New York, NY, USA, 2005. ACM Press.
- [3] V. N. Anh and A. Moffat. Pruned query evaluation using pre-computed impacts. In *SIGIR 2006*, pages 372–379, New York, NY, USA, 2006. ACM Press.
- [4] H. Bast, D. Majumdar, R. Schenkel, M. Theobald, and G. Weikum. IO-Top-k: index-access optimized top-k query processing. In *VLDB 2006*, pages 475–486. VLDB Endowment, 2006.
- [5] E. W. Brown. Fast evaluation of structured queries for information retrieval. In *SIGIR 1995*, pages 30–38, New York, NY, USA, 1995. ACM Press.
- [6] C. Buckley. Implementation of the SMART information retrieval system. Technical report, Cornell University, Ithaca, NY, USA, 1985.
- [7] S. Büttcher and C. L. A. Clarke. A document-centric approach to static index pruning in text retrieval systems. In *CIKM 2006*, pages 182–189, New York, NY, USA, 2006. ACM Press.
- [8] S. Büttcher, C. L. A. Clarke, and I. Soboroff. The TREC 2006 Terabyte track. In *TREC 2006*, Gaithersburg, Maryland USA, November 2006.
- [9] D. Carmel, D. Cohen, R. Fagin, E. Farchi, M. Herscovici, Y. S. Maarek, and A. Soffer. Static index pruning for information retrieval systems. In *SIGIR 2001*, pages 43–50, New York, NY, USA, 2001. ACM Press.
- [10] C. W. Cleverdon. The significance of the Cranfield tests on index languages. In *SIGIR 1991*, pages 3–12, New York, NY, USA, 1991. ACM Press.
- [11] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *OSDI 2004*, pages 137–150, 2004.
- [12] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. In *PODS 2001*, pages 102–113, New York, NY, USA, 2001. ACM Press.
- [13] N. Lester, A. Moffat, W. Webber, and J. Zobel. Space-limited ranked query evaluation using adaptive pruning. In *WISE 2005*, pages 470–477, 2005.
- [14] A. Moffat and J. Zobel. Self-indexing inverted files for fast text retrieval. *ACM Trans. Inf. Syst.*, 14(4):349–379, 1996.
- [15] M. Persin, J. Zobel, and R. Sacks-Davis. Filtered document retrieval with frequency-sorted indexes. *Journal of the American Society of Information Science*, 47(10):749–764, 1996.
- [16] T. Strohman, H. Turtle, and W. B. Croft. Optimization strategies for complex queries. In *SIGIR 2005*, pages 219–225, New York, NY, USA, 2005. ACM Press.
- [17] H. Turtle and J. Flood. Query evaluation: strategies and optimizations. *Information Processing and Management*, 31(6):831–850, 1995.
- [18] I. H. Witten, A. Moffat, and T. C. Bell. *Managing gigabytes (2nd ed.): compressing and indexing documents and images*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1999.
- [19] M. Zukowski, P. A. Boncz, N. Nes, and S. Heman. MonetDB/X100 - a DBMS in the CPU cache. *IEEE Data Engineering Bulletin*, 28(2):17–22, June 2005.