Space Optimizations for Total Ranking *

Douglass R. Cutting
Excite Inc.
555 Broadway, Redwood City, CA 94063

Jan O. Pedersen
Verity Inc.
894 Ross Dr., Sunnyvale, CA 94089

Abstract

Efficient ranking algorithms for similarity search use an inverted index to avoid scoring documents that have no overlap with the query. Nonetheless, partial scores must be maintained for a significant proportion of the collection. Previous work has focussed on heuristic partial ranking strategies to reduce the memory and time requirements at the cost of no longer computing the true ranks. We present two novel algorithms that efficiently compute the true total ranking with a fixed space requirement independent of the size of the collection.

*Authors listed in alphabetic order

1 Introduction

Modern information retrieval systems must scale to ever larger collections of online documents. For example, the new ARPA Tipster test collection contains over one million documents in 3.3 Gigabytes and is still growing [8]. Terrabyte collections are not uncommon in commercial settings.

Content-based access to these collections is not practicable without high performance indexing, analysis, and search algorithms. The recent NIST sponsored TREC (Text REtrieval) conferences [5, 6, 7] demonstrate that IR systems based on similarity searching with ranked output can successfully scale to these dimensions [9]. The heart of these systems is a similarity search algorithm that scores queries against documents and computes the highest scoring documents for presentation to the user in ranked order. Typical strategies employ an inverted index to avoid scoring documents that have no overlap with the query [15]. Nonetheless a significant fraction of the collection must still be scored, occupying space proportional to the size of the collection. IR researchers have investigated heuristic methods for early stopping that drop whole query terms from consideration, speeding search, and for limiting the set of documents to be scored, reducing space [1, 11, 4, 19, 16]. However, these partial ranking methods only approximate the true similarity ranking. Experiments have shown no significant loss of performance, as measured by precision and recall, due to this approximation, yet performance is sensitive to the choice of parameters, which is collection specific.

This paper will describe two new efficient algorithms for similarity search that compute a true total ranking with a fixed space requirement independent of the size of the collection. The essential idea is to reconstruct from the inverted index at query time a partial document representation for each document one-at-a-time that contains only the information necessary to compute its score with respect to the query. This is accomplished by merging together the posting from all the query terms in parallel, rather than processing them in sequence as is typical for basic similarity search algorithms.

Section 2 will present definitions, section 3 will review basic algorithms, and section 4 will describe the parallel merge algorithms.

2 Definitions

2.1 The Vector-Space Model

The *vector-space model* [18] represents a document as a high-dimensional vector with one component for each unique term in the vocabulary of the collection. The value for each component is the weight for that term in that document, often a function of the frequency of the term in that document. Terms that do not occur in the document have zero weight. Since most components will be zero, these document vectors are sparse.

Queries are also represented as vectors. *Similarity search* ranks documents with respect to their similarity to the query. In this paper similarity will be measured by the cosine of the angle between the query vector and the document vector:

$$S(d, q) = \frac{\sum_t d_t, q_t}{\|q\|\|d\|}$$

Typical weight definitions, in terms of binary document count, or the number of documents containing a term, $N(t)$, and document frequency, $f_d(t)$, are:

$$d_t = \log(f_d(t)) \log(N/N(t))$$

and

$$q_t = \log(f_q(t)) \log(N/N(t))$$

where N is the total size of the collection.

To simplify discussion, we will presume that documents weights are normalized to have unit norm. I.e.

$$d'_t = d_t/\|d\|$$

Since, for scoring purposes with respect to a particular query, $\|q\|$ is a constant, the similarity computation reduces to:

$$S(d, q) = \sum_t d'_t, q_t$$

While in principle the entire collection can be ranked, usually only the top k ranking documents are required since the user is limited in what can be conveniently viewed.

A *total ranking* computes the extract ranking of the top k documents. A *partial ranking* computes an approximate version of this total ranking and does not guarantee that the actual top k documents will be returned. Note, the query may be preprocessed to limit its size (and consequently the runtime cost) before presentation to the similarity search algorithm. For the purposes of discussion we will consider a procedure that truncates the query in this fashion but then proceeds to compute a total ranking on the remainder to be a total ranking procedure.

2.2 Indices

Document vectors are preprocessed (inverted) to create an *inverted index* that records for each term its *postings* in the collection [2]. A posting is a tuple consisting of a term, a *document id* (a unique document identifier, typically an integer), and the weight of that term in that document. The postings associated with a term are sorted by document id. Similarity search algorithms consult the postings of each query term to compute the similarity scores of documents that have terms in common with the query.

Since the size of an inverted index is proportional to the size of the collection, inverted indices are usually kept on secondary storage. The time cost of a similarity search algorithm is typically dominated by I/O time to access the inverted index. For the purposes of computing algorithm time complexity we will consider the number of disk accesses required to be proportional to the number of postings read.

2.3 Posting Streams

The postings associated with a term must be read into memory for processing. If all the postings for a term are read into memory at once, then the algorithm operates over a *postings list*, occupying space proportional to the number of postings. If, instead, the ordered postings are accessed sequentially, then the algorithm operates over a *posting stream* [3]. We will assume that access to postings through a posting stream requires only a small constant space overhead.

One implementation of a posting stream holds one disk page of postings in a fixed-size buffer. Repeated calls to the routine `next-posting` steps through all the postings, in document-id order, reading a new disk page into the buffer on demand when the buffer is exhausted.

2.4 Queues

Similarity search algorithms are required to maintain a list of the current top k scoring documents. This is conveniently implemented using a queue [10]. A queue is a data structure that maintains a partially ordered list of items. It supports `insert`, which inserts an item into the queue, `pop` which removes the current top item, with repect to the queue ordering function, from the queue, `length` which reports the current number of items in the queue, and `top` which returns the top item without modifying the queue. Through repeated calls to `pop`, the items in a queue can be extracted in sorted order. A queue can be implemented using a variety of sort-tree data structures, insuring that `pop` and `insert` require only $O(\log(k))$ comparisons, where k is the length of the queue.[1]

3 Basic Algorithms

3.1 Linear Search

The simplest similarity search algorithm does not employ an inverted index. Instead, it scans linearly through the document vectors scoring each in turn against the query vector. This algorithm is outlined in Figure 1. Note, while scanning a queue of the top k scoring documents is maintained: this avoids the cost of a final $O(N \log(N))$ sort of the document scores.

The scoring phase is linear, i.e. has cost $O(N)$, where N is the number of documents in the collection, since each document vector must be read sequentially from secondary storage. Since the cost of maintaining the top-k queue will be negligible in comparison, the overall time cost of this algorithm is linear in N. Space proportional to k is required

[1] The data structure known as the *skip list* is recommended for implementing the queues in this paper [17]. It is easy to implement, allows insertion, access and deletion to ordered sets in logarithmic time, has lower time and space constants than balanced trees and inherently allows constant time access to its least element.

```
linear_search(query) =
   queue = an empty queue of (id,score) pairs ordered by ascending score
   for (i from 1 to N)
     read d^i from disk
     score = S(q, d^i) = ∑_t d^i_t q_t
     if (length(queue)=k+1) pop(queue)
     insert((i,score),queue) ; top of queue is k+1^th item
   end
   pop(queue) ; remove k+1^th item
   return(the contents of queue in descending order)
end
```

Figure 1: Linear Search

to maintain the queue and space proportional to the largest document vector is required to buffer the document vectors while scanning. Hence the space requirements for this algorithm are roughly a constant independent of N.

3.2 Inverted-Index Search

The linear search algorithm computes many document/query similarities that are zero since many documents have no terms in common with the query. An inverted-index approach obviates these computations by only considering documents that have some overlap with the query ([12, 14] are early references; see [15] for a review).

An inverted-index similarity search algorithm processes the postings for each term in the query sequentially. Since the full score for each document is not known until all the query terms have been processed, the algorithm maintains partial scores for each document considered. As each posting is processed the partial score for its document is updated. A queue of the current top k scoring documents is maintained and returned as the result. The algorithm is outlined in Figure 2.

The largest advantage of this approach is the reduction in disk accesses. Rather than reading each document vector from secondary storage, only the p postings of the query terms need be read from the inverted index. If, as before, we neglect the cost of maintaining the top-k queue, this corresponds to an overall $O(p)$ time complexity.

Unfortunately this reduction in disk accesses comes at the cost of increased space utilization: the array of partial scores requires $O(N)$ space. One might be tempted to solve this storage problem by using a sparse data structure, such as a hash table, to store incremental scores. However, typical queries overlap a substantial fraction of the corpus (see, for example, [15]), so the overall space complexity would not be reduced. Moreover, even a space-optimized sparse data structure often requires more than two additional cells per non-zero entry, one for the id and the rest for pointers. Thus a query which overlaps one third of the collection may require *more* memory to store partial scores in a sparse

```
inverted_search(query) =
   scores = an array of length N initialized to zero
   queue = an empty queue of (id,score) pairs ordered by ascending score
   for (t ∈ q) ; iterate over terms in query
     ps = postings(t) ; a posting stream for term t
     while (p = nextposting (ps)) ; iterate over postings
       id = p.id, weight = p.weight
       scores[id] = scores[id] + q_t * weight
       if (length(queue)=k + 1) pop(queue)
       insert((id,scores[id]),queue)
     end
   end
   pop(queue)
   return(the contents of queue in descending order)
end
```

Figure 2: Inverted-Index Search

structure than in an array.

3.3 Partial Ranking

Maintaining partial scores in a sparse data structure can be profitable if one can eliminate documents from consideration. One approach is to compute an upper bound for the score of documents not yet considered [13]. For example, suppose the query terms are sorted into decreasing weight order: $q_{t_1} > q_{t_2} > \ldots > q_{t_n}$, and suppose the first m query terms have been processed. Since $\|d\| = 1$, the highest possible score for a document with current partial score zero is q_{t_j} (i.e. a document with all its weight on the next most important query term). Therefore, before processing the postings of the j^{th} query term, one can compute an upper bound, $U(j)$. If a particular posting refers to a document whose current score is zero (i.e. has no record in the sparse array of partial scores), and $U(j)$ is less than the current k^{th} top partial score, $S(k)$, then that document need not be considered. This strategy does not decrease time complexity, since all postings must still be processed, but empirical studies indicate that it may somewhat reduce space complexity [15].

It is tempting to stop early once $U(j)$ drops below $S(k)$. This offers considerable speed advantages since the query terms with small weights are often those with the most posting. However, the true ranking will not be computed. In fact, it is not even possible to guarantee that the top-k documents will be found, since the difference in score between the k^{th} and $k + 1^{\text{th}}$ top document can be small and, hence, can only be resolved by processing all the postings [19].

Nonetheless, numerous algorithms that combine document pruning with early stop-

```
partial_ranking(query) =
    scores = an empty hash table keyed by document id
    queue = an empty queue of (id,score) pairs ordered by ascending score
    for (j from 1 to n)
      compute U(j) and V(j)
      if (max_i d^i_{t_j} > V(j))
        ps = postings(t_j)
        while (p = nextposting(ps)) ; iterate over postings
        id = p.id, weight = p.weight
          if (weight > V(j))
            score = scores(id)
              if (score ≠ 0 ∨ weight > U(j))
                scores(id) = score + q_{t_j} * weight
                if (length(queue)=k + 1) pop(queue)
                insert((id,scores(id)),queue)
        end
    end
    pop(queue)
    return(the contents of queue in descending order)
end
```

Figure 3: Partial Ranking

ping have been proposed and empirically evaluated [1, 11, 4, 16]. These partial ranking algorithms are not guaranteed to return the top k documents in sorted order, but rather attempt to find k "good" documents. In practice, however, there is little measurable difference between the performance of these algorithms, as measured by precision and recall, and total ranking assuming suitable values for various parameters which may be collection specific.[2]

A partial ranking algorithm first sorts the query terms in decreasing weight order.[3] Before processing the postings of the j^{th} query term two thresholds are computed: $U(j)$ and $V(j)$, with $U(j) > V(j)$. Typically, $U(j) = \mu S(k)/q_{t_j}$ and $V(j) = \nu S(k)/q_{t_j}$ with $\nu < \mu < 1$ [16]. If $d_t < U(j)$ and d has not already been considered, then d is not added to the list of candidate documents. If $d_t < V(j)$ then the partial score for document d is not incremented. If before processing it is possible to know that $\max_i d^i_t < V(j)$ then none of the postings associated with term t need be processed. This algorithm is outlined in Figure 3.

Persin reports a reduction of a factor of fifty in space over total ranking without loss of retrieval performance given a suitable choice of μ and ν[16]. Other researchers report similar empirical results [1, 19]. Note, the asymptotic space cost is still $O(N)$ for partial

[2]Indeed, some authors report an *increase* in performance under certain conditions [16]

[3]A variation sorts query terms in decreasing $q_t \max_i d^i_t$ order [19]

ranking schemes, although the constant factor is much lower than would be the case for inverted-index search.

4 Space Efficient Total Ranking

Partial ranking addresses the large space costs of the inverted-index search algorithm by dropping potentially low scoring documents from consideration. However, this comes at the cost of introducing heuristic parameters which must be tuned for each collection [19].

The space requirement of the inverted-index algorithm arises from the need to maintain an array of partially computed scores. However, as was demonstrated by the linear algorithm, if each document score can be computed before the next document is considered, an array of partial scores need not be maintained. Recall that an inverted index speeds search by *not* processing postings for terms *not* in the query. So if *optimized* vectors containing postings only for terms in the query could be constructed sequentially for each document containing any query terms, then both the space and time advantages of the linear and inverted approaches might be had at once.

4.1 Parallel Merge

Such optimized vectors may be constructed by merging the posting streams of each of the terms in the query. Such an multi-way merge is accomplished by placing the posting streams in a queue ordered by the document id of the head posting of each stream. The top element of the queue is repeatedly removed, incremented with `next-posting`, and subsequently re-inserted in the queue, until all streams are empty. This presents the postings of all the terms in the query, sorted by document id. Runs of postings with the same document id are optimized vectors.

Figure 4 implements this algorithm. Note that it does not in fact construct optimized vectors, but rather computes the score as the postings for each document are delivered.

This algorithm computes a total ranking of the top-k most similar documents to the query, but requires only a constant space overhead and no-more disk accesses than the inverted-index search.

Memory usage includes one buffer for each term in the query, plus a queue for the top scoring documents, and thus is $O(q+k)$. Modulo constant factors, this is as small as any approach can be, as any algorithm must allocate memory for the query and results.

Each of the p postings of the query terms are processed. Hence the I/O cost of this algorithm is $O(p)$, as is the case for any inverted-index assisted total ranking algorithm. However, the CPU costs of this algorithm are higher than the inverted index search. The parallel merge algorithm performs a $\log(n)$ update to a queue of n query terms for each of p postings, and a cost $\log(k)$ update to a queue of top scoring documents for each of the $m < N$ documents containing query terms. Thus the overall CPU cost of this algorithm is $O(p \log(n) + m \log(k))$. The corresponding CPU cost for the inverted-index search is only $O(p \log(k))$. Considering that typically $p \approx m$ and $n \approx k$, this suggests that the

```
merge_search(query)
   psq = a queue of posting streams, ordered by the id of the next posting to be read
   sq = a queue of (id,score) pairs, ordered by ascending score
   insert a posting stream for each query term into psq
   while (length(psq) > 0)
     id = the id of the next posting of top(psq)
     score = 0.0
     repeat
       ps = pop(psq)
       p = nextposting(ps)
       score = score + q_p.t * p.weight
       unless (empty(ps))
         insert(ps,psq)
     until (id ≠ the id of the next posting of top(psq))
     if (length(sq)=k + 1) pop(sq)
     insert((id,score),sq)
     end
   end
   pop(sq)
   return(the contents of sq)
end
```

Figure 4: Parallel Merge Search

```
block_merge_search(query):
    scores = an array of s scores
    queue = a queue of (id,score) pairs, ordered by ascending score
    streams = a list containing one posting stream for each term in query
    for (i from 0 to N/s)
      start = i * s
      clear scores
      for (ps in streams)
        while(id of nextposting lees that start)
          p = nextposting(ps)
          index = p.id - start
          scores[index] = scores[index] + q_p.t * p.weight
          if (length(queue)=k + 1) pop(queue)
          insert((p.id,score[index]),queue)
        end
      end
    end
    pop(queue)
    return(the contents of queue in descending order)
end
```

Figure 5: Block Search

CPU costs for the parallel merge algorithm are perhaps twice as great as the inverted index search, although overall time costs in both cases are dominated by I/O.

4.2 Block Processing

The time penalty for the parallel merge as compared to the inverted index search can be mitigated by processing blocks of documents together, combining the notion of partial scoring with a bounded memory requirement.

Consider a partial score array of constant size s, say 10,000. Scores for documents 0 through
$s-1$ can be computed using a variant of the basic inverted-index algorithm which stops processing each term when a posting for a document with id greater than s is encountered. The complete scores for documents 0 through s can thus be computed in constant space. A queue of the top k is maintained as before. Next the block consisting of documents s through $2s-1$ can be processed similarly, and so on through the entire collection. This algorithm is presented in Figure 5.

The block algorithm performs the same number of disk accesses as previous inverted-index algorithms, namely $O(p)$.

The only additional memory required over the merge algorithm is that to store the array of s scores. This is a constant requirement, and hence does not affect asymptotic memory usage.

As with the original inverted-index algorithm, $O(p \log(k))$ operations are required to maintain the queue of top-scoring documents. The only additional computation required here is that of iterating through the posting streams once for each range of s documents, which adds $O(nN/s)$. This term is insignificant if we assume that p is proportional to N and that $s \gg n$.

5 Conclusion

We have presented two novel algorithms for total similarity ranking that rival inverted-index search in terms of time complexity, but reduce space complexity to optimal levels, i.e., proportional to the number of terms in the query and the number of documents requested. The first of these algorithms matches the I/O time cost of inverted-index search but requires somewhat more CPU time, while the second trades memory to mitigate that increased CPU cost.

6 Acknowledgements

Thanks to Mike Barbarino and Rick Henderson for inspiring much of this work.

References

[1] Chris Buckley and Alan F. Lewit. Optimizations of inverted vector searches. In *Proceedings of the Eighth Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 97–110, 1985.

[2] D. R. Cutting and J. O. Pedersen. Optimizations for dynamic inverted index maintenance. In *Proceedings of SIGIR'90*, September 1990. Also available as Xerox PARC technical report SSL-90-10.

[3] D.R. Cutting, J. Pedersen, and P.-K. Halvorsen. An object-oriented architecture for text retrieval. In *Conference Proceedings of RIAO'91, Intelligent Text and Image Handling, Barcelona, Spain*, pages 285–298, April 1991. Also available as Xerox PARC technical report SSL-90-83.

[4] D. Harman and G. Candela. A very fast prototype retrieval system using statistical ranking. *SIGIR Forum*, 23(3,4):100–110, Spring/Summer 1989.

[5] D.K. Harman, editor. *The First Text REtrieval Conference (TREC-1)*. NIST, 1993.

[6] D.K. Harman, editor. *The Second Text REtrieval Conference (TREC-2)*. NIST, 1994.

[7] D.K. Harman, editor. *The Third Text REtrieval Conference (TREC-3)*. NIST, 1995.

[8] Donna Harman. The TIPSTER evaluation corpus. CDROM disks of computer readable text, 1992. Available from the Linguistic Data Consortium.

[9] Donna Harman. What we have learned about document detection in tipster and trec. In *Proceedings of the TIPSTER Text Phase II 12-month meeting, Westfiled International Conference Center, Chantilly, VA*. ARPA, May 1995.

[10] D. Knuth. *The Art of Computer Programming*, volume 3: Sorting and Searching. Addison-Wesley, 1973.

[11] Dario Lucarella. A document retrieval system based on nearest neighbor searching. *Journal of Information Science*, 14:25–33, 1988.

[12] F.W. Matthews and L. Thomson. Weighted term search: a computer program for an inverted coordinate index on magnetic tape. *Journal of Chemical Documentation*, 7:49–56, 1967.

[13] J. Morrissey. An intelligent terminal for implementing relevance feedback on large operational retrieval systems. In *Proceedings of the Conference on Research and Development in Information Retrieval, Berlin*, May 1982.

[14] T. Noreault, M. Koll, and M.J. McGill. Automatic ranked output from boolean searches in sire. *JASIS*, 28:333–339, 1977.

[15] S.A. Perry and P. Willett. A review of the use of inverted files for best match searching in information retrieval systems. *Journal of Information Science*, 6:59–66, 1983.

[16] Michael Persin. Document filtering for fast ranking. In W.B. Croft and C.J. van Rijsbergen, editors, *Proceedings of the Seventeenth Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 339–349. Springer-Verlag, 1994.

[17] William Pugh. Skip lists: A probabilistic alternative to balanced trees. *Communications of the ACM*, 33(6):668–677, June 1990.

[18] G. Salton. *The SMART Retrieval System*. Prentice-Hall, Englewood Cliffs, N.J., 1971.

[19] Wai Yee Peter Wong and Dik Lun Lee. Implementations of partial document ranking using inverted files. *Information Processing and Management*, 29(5):647–669, September 1993.