Space-Limited Ranked Query Evaluation
Using Adaptive Pruning

Nicholas Lester¹, Alistair Moffat², William Webber², and Justin Zobel¹

¹ School of Computer Science and Information Technology, RMIT University, Victoria 3001, Australia
² Department of Computer Science and Software Engineering, The University of Melbourne, Victoria 3010, Australia

Abstract. Evaluation of ranked queries on large text collections can be costly in terms of processing time and memory space. Dynamic pruning techniques allow both costs to be reduced, at the potential risk of decreased retrieval effectiveness. In this paper we describe an improved query pruning mechanism that offers a more resilient tradeoff between query evaluation costs and retrieval effectiveness than do previous pruning approaches.

1 Introduction

Ranked query evaluation against a document collection requires the computation of a similarity score for each document in the collection, and then the presentation of the r highest-scoring documents, for some user-specified value r. For an introduction to the area, see Witten et al. [1999].

The most efficient way to evaluate such similarity formulations is to use a pre-computed inverted index that stores, for each term that appears in the collection, the identifiers of the documents containing it. The pointers in the postings list for a term also include any ancillary information that is required, such as the frequency of the term in the document. Term-ordered query evaluation strategies take one term at a time, and, for every document containing that term, incorporate the per-document contribution of the term into the document’s accumulator, a temporary variable created during evaluation of a query. However, in typical applications much of this computation is ultimately wasted, as is a great deal of memory. For example, in web searching tasks r = 10 answer documents are usually required, but a full set of accumulators on a collection of N = 25,000,000 documents requires perhaps 100 MB, a non-trivial amount of space.

Query pruning heuristics seek to bypass some or all of the unnecessary computation, by eliminating low-scoring documents from answer contention at a relatively early stage in the process. In this paper we examine query pruning heuristics that have been proposed in previous literature, and show that, as the size of the document collection grows, they have flaws that render them ineffective in practice. We then propose a new adaptive pruning mechanism that avoids those failings. A target number of accumulators is chosen beforehand; as each list is processed, existing low-scoring accumulators are discarded and, depending on estimates of available space that are computed adaptively, new accumulators can be added. The new mechanism allows relatively tight
control over the size of the set of accumulators, and at the same time provides a high level of agreement between the pruned ranking and the equivalent unrestricted ranking. The actual number of accumulators can be reduced to less than 1% of the number of documents without appreciable loss of effectiveness.

2 The Quit and Continue Strategies

Using a document-sorted index, the natural strategy for ranked query evaluation is to process the lists corresponding to the query terms in turn, from shortest (describing the rarest term) to longest (describing the commonest). The similarity of a given document is the sum of the contributions made by each of the query terms that occur in the document. Each term potentially contributes a partial similarity to the weight of each document in the collection; hence, because the postings lists are processed in term order rather than document order, the partial similarities need to be stored in some temporary form as the computation proceeds. As each postings list is processed, the accumulator $A_d$ is updated for each document $d$ that contains the term. Once all lists are processed, the accumulators may be normalized by the document length. Then the top $r$ values are extracted for presentation to the user.

For a query involving a common term, a large proportion of the indexed documents could be expected to end up with a non-zero accumulator, meaning that at face value the most appropriate way to store them is in an array indexed by document number $d$. However, for large collections the processing of such an array is a severe bottleneck, as its sheer size acts as an impediment to the simultaneous evaluation of multiple queries. It is therefore attractive to limit the number of accumulators in some way using a query pruning mechanism.

Probably the best-documented of the previous schemes are the quit and continue methods of Moffat and Zobel [1996]. In the quit strategy, a target number $L$ of accumulators is chosen. Initially the set $A$ of accumulators is empty. Each postings list $I_t$ is processed in turn; for each document $d \in I_t$, if $A_d$ is present in the set $A$ it is updated, otherwise a new accumulator $A_d$ is created. At the end of each list, the quit strategy checks whether the number of accumulators created so far exceeds $L$; if so, processing is terminated. We call this the full form of quit, as each list is completely processed.

A shortcoming of quit-full is that the target $L$ may be dramatically exceeded by the time the end of a list is reached. We refer to this as bursting the set of accumulators. For example suppose that $L$ is 10,000, the first list contains 1,000 entries, and the second one contains 1,000,000 entries – entirely plausible numbers in the context of web queries on web data. Then the number of accumulators created will be around 1,000,000. An alternative is the part form of quit, where the number of accumulators is checked after every posting. The accumulator target will be complied with in a strong sense, but documents with higher ordinal identifiers are much less likely to be granted an accumulator, and are thus much less likely to be returned as the answer to a query.

A greater problem is that with quit the more common terms will in many cases not contribute to the accumulators at all, leading to substantial reduction in effectiveness for smaller accumulator targets. This issue is addressed by the continue strategy. Once the target $L$ is reached, no more accumulators can be created, but processing of postings
lists continues, so that existing accumulators are updated. With continue, much smaller targets give reasonable effectiveness; Moffat and Zobel [1996] report that continue-full with $L$ set to around 1%–5% of the total number of indexed documents gives retrieval effectiveness (measured in any of the usual ways) not worse than $L = \infty$.

We report experiments with continue-full in this paper, showing, in contrast, that it is rather less successful. The difference in outcomes arises because some of the conditions assumed in the earlier work no longer apply. In particular, the queries used by Moffat and Zobel had dozens of terms (the work was completed in a universe in which web search engines did not exist, document collections were maintained by editors, and users had university degrees), which typically had a relatively broad mix of $f_t$ values, meaning that the likelihood of significant bursting was low.

The continue and quit strategies were not the first approaches used for pruning ranked query evaluation. Earlier methods include those of Smeaton and van Rijsbergen [1981], Buckley and Lewit [1985], Harman and Candela [1990], and Wong and Lee [1993]; however, these approaches are not likely to be effective in the context of current collections and system architectures. Other ranked query evaluation strategies are based on frequency-sorted indexes [Persin et al., 1996] and impact-sorted indexes [Anh et al., 2001]. These representations allow fast ranked querying, but Boolean querying becomes harder to support. Inclusion of new documents is also complex. Hence, there remain contexts in which it is appropriate to maintain postings lists in document-sorted order. It is these environments that we assume in this paper.

3 A New Approach: Adaptive Pruning

The objectives of any strategy for pruning the memory usage or computation of query evaluation are threefold: it should treat all documents in the collection equitably; it should be resistant to the “bursting” effect that was noted above; and at any given point in time it should allocate accumulators to the documents that are most likely to feature in the final ranking. Note, however, that compromise is necessary, since any scheme that processes whole postings lists before changing state risks bursting, and any scheme that makes significant state changes part way through a list cannot treat all documents equitably. The third requirement is also interesting. It suggests that a pruning mechanism must be willing to rescind as well as offer accumulators to documents.

In our proposed strategy, postings lists are again processed in decreasing order of importance, as assessed by the term weight component of the similarity heuristic. But at the commencement of processing of the list $I_t$ for term $t$, a threshold value $v_t$ is estimated, as a lower limit on an accumulator contribution that needs to be exceeded before any occurrence of $t$ is permitted to initialize an accumulator.

When a pointer and accumulator coincide, a new accumulator score is computed, by applying the update generated by that pointer. But no accumulator is retained in $A$ unless its value exceeds the current value of $v_t$, the numeric contribution that arises from (for this term) a corresponding within-document frequency hurdle of $h_t$. Thus current accumulators that do not exceed the hurdle requirement are removed from $A$ in the merge; new accumulators are created only if they have strong support indicated by $f_{d,t} \geq h_t$; and, even when a pointer updates a current accumulator, the revised value is
Algorithm 1. Processing ranked queries

Input: a set of query terms \( t \), their document frequencies \( f_t \), their collection frequencies \( F_t \), their postings lists \( I_t \), and an accumulator limit \( L \).

1: assign \( A \leftarrow \{ \} \)
2: for each term \( t \), in increasing order of \( F_t \) do
3:   use \( L, |A|, F_t \), and the previous threshold \( v_t \) to establish a new threshold \( v_t \)
4:   for each document \( d \) in \( A \cup I_t \) do
5:     if \( d \in I_t \) then
6:       calculate a contribution \( c \) using \( F_t \) and \( f_{d,t} \)
7:     else
8:       assign \( c \leftarrow 0 \)
9:     if \( d \in A \) then
10:       assign \( c \leftarrow A_d + c \)
11:     if \( c \geq v_t \) then
12:       assign \( A_d \leftarrow c \) and \( A \leftarrow A \cup \{A_d\} \)
13:     else if \( d \in A \) then
14:       assign \( A \leftarrow A \setminus \{A_d\} \)
15:     pause periodically to reevaluate \( v_t \), tracking the current size of \( A \), and the rate at which it has been changing relative to the target rate of change

Output: a set of approximately \( L \) accumulator values, not yet normalized by document length

allowed to stay as a candidate only if it exceeds the current \( v_t \). That is, the combination of \( v_t \) (as a similarity score) and \( h_t \) (as a corresponding \( f_{d,t} \) threshold) act as a “scraper”, that removes low-value candidates from \( A \) and replaces them by any new high-valued accumulators represented by the current list.

Algorithm 1 summarizes this process. The set of accumulators \( A \) and \( I_t \) are both lists sorted by document number, and are jointly processed in a merge-like operation.

The critical component is that of setting, for each term’s list \( I_t \), a minimum value \( h_t \) on \( f_{d,t} \) scores that are to be allowed to create an accumulator. The threshold \( h_t \) should be set so that it changes as little as possible during the processing of \( I_t \) (the equity principle); and so that at the end of \( I_t \), the number of accumulators is reasonably close to target \( L \) (the principle of using the available accumulators wisely). Setting \( h_t \) (and hence \( v_t \), which is directly related to \( h_t \)) too low means that too many new accumulators will get created from term \( t \), and not enough old ones get reclaimed, making the total number of accumulators grow beyond \( L \). On the other hand, setting \( h_t \) too high means that plausible candidates from \( I_t \) get denied, and equally plausible candidates from \( A \) get unnecessarily discarded, resulting in a pool of fewer than \( L \) candidates being made available to the final ranking process. Another way of looking at \( v_t \) is that as far as possible it should be set at the beginning of the processing of \( I_t \) to a value that would be equal, were no pruning at all taking place, to the \( L \)th largest of a complete set of accumulators at the end of processing \( I_t \).

At the beginning of processing, if a term cannot possibly cause the accumulator limit to be exceeded because \( |A| + f_t \leq L \), then \( h_t \) is set to one, and every document in \( I_t \) without an accumulator is allocated one.

Once the accumulator target is under threat by a term for which \( |A| + f_t > L \) (potentially even the first query term, if \( f_t > L \)), a larger \( h_t \) is derived. If the entries
Algorithm 2. Adaptively estimate the thresholding parameter for a term

Inputs: a set of accumulators $A$, a term $t$, and an accumulator target $L$

1: assign $startA \leftarrow |A|$
2: assign $p \leftarrow f_t / L$
3: if this is the first term that risks exceeding $L$ accumulators then
   4: assign $h_t \leftarrow \max\{f_{d,t} \mid d \in \text{the first } p \text{ pointers in } I_t\}$
5: else
   6: assign $h_t$ to be the $f_{d,t}$ frequency corresponding to the previous value of $v_t$
7: assign $s \leftarrow h_t / 2$
8: while pointers remain in $I_t$ do
   9: process pointers through until the $p$th as described in Algorithm 1, using $h_t$ as a term frequency threshold, and the corresponding $v_t$ as an accumulator value threshold
10: assign $predict \leftarrow |A| + (f_t - p) \times (|A| - startA) / p$
11: if $predict > \theta L$ then
   12: assign $h_t \leftarrow h_t + s$ and recalculate $v_t$
13: else if $predict < L / \theta$ then
   14: assign $h_t \leftarrow h_t - s$ and recalculate $v_t$
15: assign $p \leftarrow 2p + 1$
16: assign $s \leftarrow (s + 1) / 2$

in each list $I_t$ were homogeneous, a sampling approach could be used to select $h_t$, and that value could be used for the whole list. Unfortunately, postings lists are rarely homogeneous – term usage can change dramatically from one end of a collection to another, and factors such as document length can also have a significant bearing.

Algorithm 2 shows an adaptive estimation process that addresses these problems. The philosophy is that it is acceptable to adjust $h_t$ as each list is being processed, provided that the bulk of each list is handled using values that do not differ too much; and that, conversely, if big shifts in $h_t$ become necessary, they should be made in such a way that as few as possible of the list’s pointers are treated unfairly. The objective is to end the processing of this term with $L$ accumulators. However it is impossible to hit such a target exactly, so a tolerance $\theta$ is allowed, and any arrangement in which $L / \theta \leq |A| \leq \theta L$ is tolerable. A typical value might be $\theta = 1.2$.

In Algorithm 2 the first step is to record, using variable $startA$, the number of accumulators at the commencement of processing this term. After some number $p$ of the $f_t$ pointers in $I_t$ have been processed, the change in the size of the set $A$ is clearly $|A| - startA$. Extrapolating forwards over the remaining $f_t - p$ pointers in $I_t$ allows calculation of a quantity $predict$, the number of accumulators expected if the remainder of the list $I_t$ is homogeneous with respect to the part already processed. If $predict$ is higher than $\theta L$, it is time to increase $h_t$ and $v_t$ so as to reduce the rate at which $A$ is growing; and, if $predict$ is less than $L / \theta$, then $h_t$ and $v_t$ should be decreased.

Increases and decreases to $h_t$ are by an amount $s$, which is initially large relative to $h_t$, but halves at each evaluation until it reaches one. At the same time, the intervals $p$ over which $h_t$ is held constant are doubled after each reevaluation. The initial interval $p$ is set to $f_t / L$, namely, the interval over which (if $I_t$ were homogeneous) one accumulator might expect to have been identified. Taking $h_t$ to be the maximum $f_{d,t}$ identified in the first $p$ pointers is thus a plausible initial estimate.
4 Evaluation Methodology

The standard method for evaluating a retrieval mechanism is via a controlled text collection, a set of queries, and a set of partial or full relevance judgments that indicate which of the documents are relevant to which query. For example the NIST TREC project has created several such resources; see trec.nist.gov. In addition, a range of effectiveness metrics have been defined, with mean average precision (MAP) perhaps the most widely used [Buckley and Voorhees, 2000]. In the experiments here we make use of the 426 GB TREC GOV2 collection, which contains approximately 25 million web documents; and the short queries associated with query topics 701–750. In terms of scale, our experimentation is thus realistic of whole-of-web searching using a document distributed retrieval system on a cluster of 100 computers [Barroso et al., 2003].

Another interesting question that arises is how to count accumulators, and what type of averaging is appropriate over a query stream. The obvious possibilities would be to take the absolute maximum requirement over the stream; or to take the average of the per-query maximums. But both of these have drawbacks, and do not accurately reflect what it is that we wish to quantify, namely, the amount of memory in a parallel query handling system that is required on average by each query thread. Instead, we measure the time-averaged accumulator requirement over the query stream, by tracking the number of active accumulators throughout the processing, and disregarding query boundaries. Queries that have a high accumulator load for an extended period are then accurately counted, since they contribute through more of the time intervals.

5 Experiments

Table 1 shows the result of applying the two variants of the continue strategy to web-scale data using short web-like queries, and compares their performance to the adaptive

Table 1. Retrieval effectiveness scores, using TREC topics 701–750 (short queries) the GOV2 collection, retrieval depth \( r = 1,000 \), and a language model with Dirichlet smoothing [Zhai and Lafferty, 2004]. Numbers reported are mean average precision (MAP), and the time-averaged number of accumulators required to process the query stream. A full evaluation of each query leads to a MAP of 0.240.

<table>
<thead>
<tr>
<th>Target (’000)</th>
<th>Target (’000)</th>
<th>Actual (’000)</th>
<th>MAP</th>
<th>Actual (’000)</th>
<th>MAP</th>
<th>Actual (’000)</th>
<th>MAP</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.0</td>
<td>0.045</td>
<td>237.9</td>
<td>0.235</td>
<td>1.5</td>
<td>0.150</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>2.0</td>
<td>0.065</td>
<td>238.9</td>
<td>0.235</td>
<td>3.2</td>
<td>0.179</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>4.0</td>
<td>0.093</td>
<td>252.4</td>
<td>0.235</td>
<td>5.8</td>
<td>0.202</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>10.0</td>
<td>0.126</td>
<td>260.3</td>
<td>0.235</td>
<td>13.0</td>
<td>0.215</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>19.9</td>
<td>0.142</td>
<td>372.1</td>
<td>0.235</td>
<td>26.5</td>
<td>0.228</td>
<td></td>
</tr>
<tr>
<td>40</td>
<td>39.8</td>
<td>0.141</td>
<td>478.7</td>
<td>0.235</td>
<td>47.7</td>
<td>0.233</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>98.5</td>
<td>0.170</td>
<td>533.7</td>
<td>0.235</td>
<td>121.1</td>
<td>0.237</td>
<td></td>
</tr>
<tr>
<td>200</td>
<td>194.1</td>
<td>0.194</td>
<td>599.9</td>
<td>0.237</td>
<td>214.7</td>
<td>0.239</td>
<td></td>
</tr>
<tr>
<td>400</td>
<td>373.8</td>
<td>0.212</td>
<td>1,590.6</td>
<td>0.239</td>
<td>395.5</td>
<td>0.240</td>
<td></td>
</tr>
<tr>
<td>1,000</td>
<td>845.3</td>
<td>0.221</td>
<td>2,862.0</td>
<td>0.240</td>
<td>900.0</td>
<td>0.240</td>
<td></td>
</tr>
</tbody>
</table>
The control knob in these experiments is the target number of accumulators, shown in the first column. The next three pairs of columns show respectively the continue-part mechanism, the continue-full mechanism, and the new method. The time-averaged number of accumulators used for the continue-part combination is always within the target (and less than the target in some cases because some queries cannot support even 20,000 accumulators), so in that sense part is a useful technique. But the loss of retrieval effectiveness compared to full evaluation is acute. We also experimented with the two quit versions, and confirmed that they are faster, but yielded even worse effectiveness scores for the same level of accumulator usage.

The next pair of columns demonstrate why the retrieval is so bad with part – the actual number of accumulators required just to equitably complete the processing of the boundary term is enormously bigger than the accumulator target. That is, in the part strategy only a small fraction of the boundary term gets processed before the target is reached, and documents early in the collection are greatly favored. On the other hand, fully processing the boundary term can give good retrieval effectiveness – especially with the continue strategy – but causes the accumulator set to burst.

The final pair of columns show the new method. In all cases the time-averaged accumulator requirement is close to the target value $L$. More interesting is the retrieval performance – with as few as 100,000 accumulators, just 0.4% of the size of the collection, very good MAP results are obtained.

Figure 1 shows the same data graphically, but with average memory space presented as a ratio of accumulators to documents in the collection. The two variants of the continue approach provide bookends to performance, with the full version requiring very large numbers of accumulators, irrespective of the target $L$, but providing high effectiveness; and the part version tightly bounding the number of accumulators, but sacrificing retrieval effectiveness as measured by MAP.

6 Conclusions

The adaptive pruning method presented here combines the strengths of the two continue approaches and eliminates their weaknesses, in that the number of accumulators can be
reasonably controlled to a specified target, and for a given accumulator consumption, retrieval effectiveness is excellent. Adaptive query pruning is a useful technique that allows memory usage to be tightly controlled, even when carrying out search operations on web-scale document collections.

Acknowledgment. This work was supported by the Australian Research Council.

References


