5 Query Processing

In Chapter 4 we explored the fundamental data structures that make up an inverted index. We now discuss how to realize efficient search operations on top of these data structures. Details of the search process and its implementation can vary from search engine to search engine. However, the essential ideas and algorithms are usually the same, regardless of whether we are dealing with a single-user desktop search system or a large-scale Web search engine.

The most basic retrieval model is quite likely the Boolean model that we have already discussed in Section 2.2. In the Boolean model, each term represents the set of documents in which it appears. Document sets may be combined by using the standard operators AND (set intersection), OR (union), and NOT (inversion). In this chapter we explore two popular alternatives to the Boolean model. The first one, ranked retrieval (Section 5.1), allows the search engine to rank search results according to their predicted relevance to the query. The second one, lightweight structure (Section 5.2), is a natural extension of the Boolean model to the sub-document level. Instead of restricting the search process to entire documents, it allows the user to search for arbitrary text passages satisfying Boolean-like constraints (e.g., "show me all passages that contain 'apothecary' and 'drugs' within 10 words").

Although we will occasionally refer to effectiveness measures, such as mean average precision (MAP), the focus of this chapter is not so much on the quality of the search results as on the efficiency of the search process. Quality aspects of various retrieval functions are covered in Chapters 8 and 9.

5.1 Query Processing for Ranked Retrieval

As pointed out in Section 2.2, Boolean retrieval and ranked retrieval are not mutually exclusive, but complementary: The search engine may use a Boolean interpretation of the user's query to determine the set of matching documents. For instance, given the query

$$Q = \langle \text{“greek”}, \text{“philosophy”}, \text{“stoicism”} \rangle \tag{5.1}$$

(TREC topic 433), the search engine may retrieve all documents that match the *conjunctive* Boolean query

$$\text{“greek” AND “philosophy” AND “stoicism”} \tag{5.2}$$

or the *disjunctive* Boolean query

$$\text{“greek” OR “philosophy” OR “stoicism”} \tag{5.3}$$

and then rank those documents according to their similarity to Q, for example, using the cosine measure (Equation 2.12 on page 56).

Traditional information retrieval systems usually follow the disjunctive approach, while Web search engines often employ conjunctive query semantics. The conjunctive retrieval model leads to faster query processing than the disjunctive model, because fewer documents have to be scored and ranked. However, this performance advantage comes at the cost of a lower recall: If a relevant document contains only two of the three query terms, it will never be returned to the user. This limitation is quite obvious for the query Q shown above. Of the half-million documents in the TREC collection, 7,834 match the disjunctive interpretation of the query, whereas only a single document matches the conjunctive version. Incidentally, that document is not even relevant (it is about a Mexican actor and his latest film projects).

The failure of the conjunctive approach for the above query can be explained as follows: Authors familiar with the concept of Stoicism do not feel the need to mention that it is a philosophy or that it originated in Greece; both details seem obvious to them. The user might think that she helps the search engine by providing the additional keywords "philosophy" and "greek", when in fact adding these terms makes the search results worse. This effect becomes even more pronounced for longer queries, in which the presence of a single unwisely chosen (or misspelled) query term can completely ruin the search results.

We hold the view that adding relevant terms to the query should not hurt the search results. Throughout the following sections, we assume that the search engine follows the disjunctive approach, retrieving all documents that contain at least one of the query terms, and leaving it to the ranking function to determine which of these documents best match the user's query. Of course, there is no doubt that the conjunctive model usually allows more efficient query processing than the disjunctive model. Therefore, many of the optimizations covered in this section share the same common goal: to narrow the performance gap between the AND interpretation and the OR interpretation of a given query.

Okapi BM25

For the purpose of our discussion, we assume that the search engine employs the Okapi BM25 scoring function (Equation 8.48 on page 272) to rank documents based on their predicted relevance to the query. For convenience, we repeat the ranking formula here:

$$\text{Score}_{\text{BM25}}(q,d) = \sum_{t \in q} \log\left(\frac{N}{N_t}\right) \cdot \text{TF}_{\text{BM25}}(t,d), \tag{5.4}$$

$$\text{TF}_{\text{BM25}}(t,d) = \frac{f_{t,d} \cdot (k_1 + 1)}{f_{t,d} + k_1 \cdot ((1-b) + b \cdot (l_d / l_{avg}))}. \tag{5.5}$$

The formula has two free parameters: k_1 (default value: $k_1 = 1.2$), which regulates how fast the TF component saturates, and b (default value: $b = 0.75$), which controls the degree of

document length normalization. All other variables have the standard semantics, as listed in the "Notation" table at the beginning of this book.

If you are interested in the theory underlying the BM25 formula, you can find a derivation and in-depth discussion in Chapter 8. In order to follow the material covered in this section, a full understanding of BM25 is not required. We do, however, want to focus briefly on the role of the parameter k_1, as it interacts closely with various query optimizations for ranked retrieval. k_1 caps the score contribution of an individual query term:

$$\lim_{f_{t,d} \to \infty} \mathrm{TF}_{\mathrm{BM25}}(t, d) \;=\; k_1 + 1. \tag{5.6}$$

For the default value $k_1 = 1.2$, the TF contribution of any given query term can never exceed 2.2. Because of this rather tight upper bound, a document that contains two different query terms is far more likely to be ranked highly than a document that contains only a single query term, even if the latter contains that term many times.

Suppose we are given the query $q = \langle t_1, t_2 \rangle$, with $N_{t_1} \approx N_{t_2}$, and we want to score two documents d_1 and d_2 of average length (i.e., $l_{d_1} = l_{d_2} = l_{avg}$). Further suppose that d_1 contains one occurrence of each t_1 and t_2, while d_2 contains 10 occurrences of t_1 and no occurrence of t_2. Then we have:

$$\mathrm{Score}_{\mathrm{BM25}}(q, d_1) \;\approx\; \log\left(\frac{N}{N_{t_1}}\right) \cdot \left(2 \cdot \frac{1 \cdot (k_1 + 1)}{1 + k_1}\right) \;\approx\; 2 \cdot \log\left(\frac{N}{N_{t_1}}\right), \tag{5.7}$$

$$\mathrm{Score}_{\mathrm{BM25}}(q, d_2) \;\approx\; \log\left(\frac{N}{N_{t_1}}\right) \cdot \frac{10 \cdot (k_1 + 1)}{10 + k_1} \;\approx\; 1.95 \cdot \log\left(\frac{N}{N_{t_1}}\right) \tag{5.8}$$

(for $k_1 = 1.2$). Later in this section we will use the upper bounds provided by k_1 to ignore postings for which we know a priori that they cannot push the corresponding document into the top search results.

5.1.1 Document-at-a-Time Query Processing

The most common form of query processing for ranked retrieval is called the *document-at-a-time* approach. In this method all matching documents are enumerated, one after the other, and a score is computed for each of them. At the end all documents are sorted according to their score, and the top k results (where k is chosen by the user or the application) are returned to the user.

Figure 5.1 shows a document-at-a-time algorithm for BM25. The algorithm — except for the score computation — is the same as the **rankCosine** algorithm in Figure 2.9 (page 59). The overall time complexity of the algorithm is

$$\Theta(m \cdot n + m \cdot \log(m)), \tag{5.9}$$

```
    rankBM25_DocumentAtATime (⟨t_1, ..., t_n⟩, k) ≡
1     m ← 0   // m is the total number of matching documents
2     d ← min_{1≤i≤n} {nextDoc(t_i, −∞)}
3     while d < ∞ do
4        results[m].docid ← d
5        results[m].score ← Σ_{i=1}^{n} log(N/N_{t_i}) · TF_BM25(t_i, d)
6        m ← m + 1
7        d ← min_{1≤i≤n} {nextDoc(t_i, d)}
8     sort results[0..(m − 1)] in decreasing order of score
9     return results[0..(k − 1)]
```

Figure 5.1 Document-at-a-time query processing with BM25.

where n is the number of query terms and m is the number of matching documents (containing at least one query term). The term $m \cdot n$ corresponds to the loop starting in line 3 of the algorithm. The term $m \cdot \log(m)$ corresponds to the sorting of the search results in line 8.

It is sometimes difficult to use Equation 5.9 directly, because the value of m is not known without running the algorithm and enumerating all search results. Depending on how often the query terms co-occur in a document, m can be anywhere between N_q/n and N_q, where $N_q = N_{t_1} + \cdots + N_{t_n}$ is the total number of postings of all query terms. In the worst case, each matching document contains only a single query term. The time complexity of the algorithm in Figure 5.1 is then

$$\Theta(N_q \cdot n + N_q \cdot \log(N_q)). \tag{5.10}$$

In practice the vast majority of all matching documents do in fact contain only one query term. For instance, for our earlier example query ⟨"greek", "philosophy", "stoicism"⟩ we have $m = 7835$ and $N_q = 7921$. Thus, Equation 5.10 is usually a good approximation of Equation 5.9.

The basic document-at-a-time algorithm shown in Figure 5.1 has two major sources of inefficiency:

- The computations carried out in lines 5 and 7 need to iterate over all n query terms, regardless of whether they actually appear in the current document or not. If n is large, then this can become a problem. Consider the extreme case in which we have ten query terms, and each matching document contains exactly one of them. Then the algorithm performs ten times as much work as it would if it processed each term individually.
- The final sorting step in line 8 sorts all documents in the *results* array. Of course, sorting the whole array is a waste of resources, given that we are interested in only the top k results. The $\Theta(m \cdot \log(m))$ complexity of the sorting process might not seem so bad. But keep in mind that we are potentially dealing with a few million matching documents, so the $m \cdot \log(m)$ term in Equation 5.9 may in fact outweigh the $m \cdot n$ term corresponding to the score computation.

We can address both problems in the same way: by employing a data structure known as *heap*. If you already know what a heap is, feel free to skip ahead to the section "Efficient query processing with heaps" (page 142).

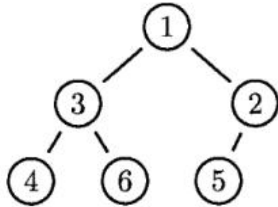
Binary heaps

A heap (or, more precisely, a *binary min-heap*) is a binary tree that satisfies the following definition:

1. Every empty binary tree is a heap.
2. A nonempty binary tree $\mathcal{T}$ is a heap if all of the following hold:
   (a) $\mathcal{T}$ is completely filled on all levels, except possibly the deepest one.
   (b) $\mathcal{T}$'s deepest level is filled from left to right.
   (c) For each node v, the value stored in v is smaller than the values stored in any of v's children.

Requirements (a) and (b), in combination, are sometimes referred to as the *shape property*. Because of the shape property, a heap may be represented not only as a tree but also as an array. The tree's root node is then stored in position 0, and the children for a node in position i are stored in position $2i+1$ and $2i+2$, respectively. Figure 5.2(a) shows the tree representation of a heap for the set $\{1,2,3,4,5,6\}$. Figure 5.2(b) shows the equivalent array representation.

(a) Tree representation



(b) Array representation

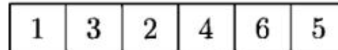


Figure 5.2 A heap for $\{1,2,3,4,5,6\}$, in tree representation and equivalent array representation.

In practical implementations the array representation is almost always preferred over the tree representation because it is more space efficient (no child pointers) and also faster (better data locality). In the array representation, requirement 2(c) from above translates into

$$\begin{aligned}&\forall i \in \{0,1,2,\ldots,len-1\}:\\&\quad (2i+1 \geq len \;\vee\; A[i] \leq A[2i+1]) \;\wedge\; (2i+2 \geq len \;\vee\; A[i] \leq A[2i+2]),\end{aligned} \tag{5.11}$$

where *len* is the length of the array A. In particular, every sorted array is a heap (but not every heap is a sorted array).

Heaps support a variety of interesting operations. The only operation we are interested in here is called REHEAP. Consider the heap from Figure 5.2. Suppose we want to replace the root's value, 1, with a new value, say 9, while maintaining the heap property. The REHEAP algorithm would do this by replacing 1 with 9, then swapping 9 with 2 and finally swapping 9 with 5, resulting in the new array $\langle 2, 3, 5, 4, 6, 9 \rangle$, which again has the heap property. More formally, REHEAP takes a binary tree that meets all requirements for being a heap except for 2(c), and transforms it into a heap by moving the root's value down the tree until the heap property is established.

What is the time complexity of this algorithm? Consider a heap $\mathcal{T}$ containing n elements. Because every heap is a balanced binary tree, we know that each of $\mathcal{T}$'s leaves has either height $\lfloor \log_2(n) \rfloor$ or height $\lfloor \log_2(n) - 1 \rfloor$. Therefore, REHEAP will finish in $O(\log(n))$ steps.

Efficient query processing with heaps

We can use REHEAP to overcome the limitations of the algorithm in Figure 5.1. In the revised version of the algorithm, we employ two heaps: one to manage the query terms and, for each term t, keep track of the next document that contains t; the other one to maintain the set of the top k search results seen so far.

The resulting algorithm is shown in Figure 5.3. The *terms* heap contains the set of query terms, ordered by the next document in which the respective term appears (*nextDoc*). It allows us to perform an efficient multiway merge operation on the n postings lists. The *results* heap contains the top k documents encountered so far, ordered by their scores. It is important to note that *results*'s root node (i.e., *results*[0] in the array representation employed by the algorithm) does not contain the best document seen so far, but the kth-best document seen so far. This allows us to maintain and continually update the top k search results by replacing the lowest-scoring document in the top k (and restoring the heap property) whenever we find a new document that scores better than the old one.

The worst-case time complexity of the revised version of the document-at-a-time algorithm is

$$\Theta(N_q \cdot \log(n) + N_q \cdot \log(k)), \quad (5.12)$$

where $N_q = N_{t_1} + \cdots + N_{t_n}$ is the total number of postings for all query terms. The first term $(N_q \cdot \log(n))$ corresponds to REHEAP operations carried out on the *terms* heap, restoring the heap property after every posting. The second term $(N_q \cdot \log(k))$ corresponds to REHEAP operations carried out on the *results* heap, restoring the heap property whenever a new document is added to the set of top k results.

Compared to Equation 5.10, the revised algorithm constitutes a substantial improvement, primarily because of the speedup achieved by restricting the final sorting procedure to the top k documents instead of all matching documents. Moreover, although maintaining the top-k set has indeed a worst-case complexity of $\Theta(N_q \cdot \log(k))$, this worst-case complexity plays no role in practice; the algorithm's average-case complexity is even better than what is shown in Equation 5.12 (see Exercise 5.1).

```
    rankBM25_DocumentAtATime_WithHeaps (⟨t_1, ..., t_n⟩, k) ≡
1       for i ← 1 to k do   // create a min-heap for the top k search results
2           results[i].score ← 0
3       for i ← 1 to n do   // create a min-heap for the n query terms
4           terms[i].term ← t_i
5           terms[i].nextDoc ← nextDoc(t_i, −∞)
6       sort terms in increasing order of nextDoc   // establish heap property for terms
7       while terms[0].nextDoc < ∞ do
8           d ← terms[0].nextDoc
9           score ← 0
10          while terms[0].nextDoc = d do
11              t ← terms[0].term
12              score ← score + log(N/N_t) · TF_BM25(t, d)
13              terms[0].nextDoc ← nextDoc(t, d)
14              reheap(terms)   // restore heap property for terms
15          if score > results[0].score then
16              results[0].docid ← d
17              results[0].score ← score
18              reheap(results)   // restore heap property for results
19      remove from results all items with score = 0
20      sort results in decreasing order of score
21      return results
```

Figure 5.3 Document-at-a-time query processing with BM25, using binary heaps for managing the set of terms and managing the set of top-k documents.

MaxScore

Although the algorithm in Figure 5.3 already achieves reasonable query processing performance, there is still room for improvement. Recall from our earlier discussion that the BM25 TF contribution can never exceed $k_1 + 1 = 2.2$. Thus, the overall score contribution of a term t is bounded from above by $2.2 \cdot \log(N/N_t)$. This bound is called the term's *MaxScore*. Now consider again the query

$$Q = \langle \text{“greek”}, \text{“philosophy”}, \text{“stoicism”} \rangle . \tag{5.13}$$

The query terms' document frequencies, IDF weights, and corresponding MaxScore values are:

Term	N_t	$\log_2(N/N_t)$	MaxScore
“greek”	4,504	6.874	15.123
“philosophy”	3,359	7.297	16.053
“stoicism”	58	13.153	28.936

Table 5.1 Total time per query and CPU time per query, with and without MAXSCORE. Data set: 10,000 queries from TREC TB 2006, evaluated against a frequency index for GOV2.

	Without MaxScore			With MaxScore		
	Wall Time	**CPU**	**Docs Scored**	**Wall Time**	**CPU**	**Docs Scored**
OR, k=10	400 ms	304 ms	$4.4 \cdot 10^6$	188 ms	93 ms	$2.8 \cdot 10^5$
OR, k=100	402 ms	306 ms	$4.4 \cdot 10^6$	206 ms	110 ms	$3.9 \cdot 10^5$
OR, k=1000	426 ms	329 ms	$4.4 \cdot 10^6$	249 ms	152 ms	$6.2 \cdot 10^5$
AND, k=10	160 ms	62 ms	$2.8 \cdot 10^4$	n/a	n/a	n/a

Suppose the user who entered the query is interested in the top $k = 10$ search results. After scoring a few hundred documents, we may encounter the situation in which the 10th-best result found so far exceeds the maximum score contribution of the term "greek". That is,

$$\textit{results}[0].\textit{score} > \text{MaxScore}(\text{"greek"}) = 15.123. \tag{5.14}$$

When this happens, we know that a document that only contains "greek", but neither "philosophy" nor "stoicism", can never make it into the top 10 search results. Thus, there is no need even to score any documents that contain only "greek". We may remove the term from the *terms* heap and look at its postings only when we find a document that contains one of the other two query terms.

As we compute scores for more and more documents, we may at some point encounter the situation in which

$$\textit{results}[0].\textit{score} > \text{MaxScore}(\text{"greek"}) + \text{MaxScore}(\text{"philosophy"}) = 31.176. \tag{5.15}$$

At that point, we know that a document can make it into the top 10 only if it contains the term "stoicism", and we can remove "philosophy" from the *terms* heap, just as we did for "greek".

The strategy outlined above is called MAXSCORE. It is due to Turtle and Flood (1995). MAXSCORE is guaranteed to produce the same set of top k results as the algorithm in Figure 5.3, but to do so much faster because it ignores all documents for which we know a priori that they cannot be part of the final top k.

Note that, even though MAXSCORE removes some terms from the heap, it still uses them for scoring. This is done by maintaining two data structures, one for terms that are still on the heap and the other for terms that have been removed from the heap. Whenever we find a document d that contains one of the terms still on the heap, we iterate over the set of terms removed from the heap, and for each term t we call **nextDoc**$(t, d - 1)$ to determine whether t appears in d. If it does, we compute t's score contribution and add it to the score for d.

Table 5.1 lists the average time per query and average CPU time per query for 10,000 queries from the TREC 2006 Terabyte track, run against an on-disk frequency index for the GOV2

collection (containing postings of the form $(d, f_{t,d})$). For $k = 10$, the default setting of many search engines, MaxScore decreases the total time per query by 53% and the CPU time per query by 69%, compared to the algorithm in Figure 5.3. It does this by reducing the total number of documents scored by the search engine, from 4.4 million to 280,000 documents per query on average. Note that, even with MaxScore, the search engine still has to score an order of magnitude more documents than it would if it employed a conjunctive retrieval model (Boolean AND). However, in terms of overall performance, Boolean OR with MaxScore is not very far from Boolean AND. The average CPU time per query, for instance, is only 50% higher: 93 ms per query versus 62 ms.

5.1.2 Term-at-a-Time Query Processing

As an alternative to the document-at-a-time approach, some search engines process queries in a *term-at-a-time* fashion. Instead of merging the query terms' postings lists by using a heap, the search engine examines, in turn, all (or some) of the postings for each query term. It maintains a set of document score *accumulators*. For each posting inspected, it identifies the corresponding accumulator and updates its value according to the posting's score contribution to the respective document. When all query terms have been processed, the accumulators contain the final scores of all matching documents, and a heap may be used to collect the top k search results.

One of the motivations behind the term-at-a-time approach is that the index is stored on disk and that the query terms' postings lists may be too large to be loaded into memory in their entirety. In that situation a document-at-a-time implementation would need to jump back and forth between the query terms' postings lists, reading a small number of postings into memory after each such jump, and incurring the cost of a nonsequential disk access (disk seek). For short queries, containing two or three terms, this may not be a problem, as we can keep the number of disk seeks low by allocating an appropriately sized read-ahead buffer for each postings list. However, for queries containing more than a dozen terms (e.g., after applying pseudo-relevance feedback — see Section 8.6), disk seeks may become a problem. A term-at-a-time implementation does not exhibit any nonsequential disk access pattern. The search engine processes each term's postings list in a linear fashion, moving on to term t_{i+1} when it is done with term t_i.

Because the term-at-a-time paradigm processes each postings list separately, it is typically used only for scoring functions that are of the form

$$\text{score}(q, d) \;=\; \text{quality}(d) + \sum_{t \in q} \text{score}(t, d)\,. \tag{5.16}$$

In this equation, quality(d) is an optional, query-independent score component, for example, PageRank (Equation 15.8 on page 518). Most traditional scoring functions, including VSM (Equation 2.12 on page 56), BM25 (Equation 8.48 on page 272) and LMD (Equation 9.32 on page 295), are of this (or an equivalent) form. They are sometimes referred to as *bag-of-words*

```
    rankBM25_TermAtATime (⟨t_1, ..., t_n⟩, k) ≡
1       sort ⟨t_1, ..., t_n⟩ in increasing order of N_{t_i}
2       acc ← {}, acc' ← {}   // initialize two empty accumulator sets
3       acc[0].docid ← ∞   // end-of-list marker
4       for i ← 1 to n do
5           inPos ← 0   // current position in acc
6           outPos ← 0   // current position in acc'
7           for each document d in t_i's postings list do
8               while acc[inPos] < d do   // copy accumulators from acc to acc'
9                   acc'[outPos++] ← acc[inPos++]
10              acc'[outPos].docid ← d
11              acc'[outPos].score ← log(N/N_{t_i}) · TF_BM25(t_i, d)
12              if acc[inPos].docid = d then   // term and accumulator coincide
13                  acc'[outPos].score ← acc'[outPos].score + acc[inPos++].score
14              d ← nextDoc(t_i, acc'[outPos])
15              outPos ← outPos + 1
16          while acc[inPos] < ∞ do   // copy remaining accumulators from acc to acc'
17              acc'[outPos++] ← acc[inPos++]
18          acc'[outPos].docid ← ∞   // end-of-list marker
19          swap acc and acc'
20      return the top k items of acc   // use a heap to select the top k
```

Figure 5.4 Term-at-a-time query processing with BM25. Document scores are stored in accumulators. The accumulator array is traversed co-sequentially with the current term's postings list.

methods. Scoring functions that take into account the proximity of query terms in the document, including phrase queries, are incompatible with Equation 5.16. In theory, they may still be implemented within a term-at-a-time query processing framework. However, the per-document accumulators maintained by the search engine would need to hold some extra information (e.g., term positions) in addition to the documents' scores, thus increasing their size substantially.

Figure 5.4 shows a possible term-at-a-time query processing algorithm for BM25. In the figure, the score accumulators are kept in the array *acc*. For each query term t_i, the array is traversed co-sequentially with t_i's postings list, creating a new array *acc'* that contains the updated accumulators. The worst-case time complexity of this algorithm is

$$\Theta\left(\sum_{i=1}^{n}(N_q/n \cdot i) + N_q \cdot \log(k)\right) \;=\; \Theta(N_q \cdot n + N_q \cdot \log(k)), \tag{5.17}$$

where $N_q = N_{t_1} + \cdots + N_{t_n}$ is the total number of postings for all query terms. The worst case occurs when $N_{t_i} = N_q/n$ for $1 \leq i \leq n$.

By comparing Equation 5.17 with Equation 5.12, we see that the term-at-a-time algorithm, at least in the form presented in Figure 5.4, is actually slightly slower than the document-at-a-time algorithm ($N_q \cdot n$ instead of $N_q \cdot \log(n)$), due to the necessity to traverse the entire accumulator

set for every query term t_i. In theory, we could eliminate this bottleneck by replacing the array implementation for *acc* with a hash table so that each accumulator update is an $O(1)$ operation. This would result in an overall complexity of $\Theta(N_q + N_q \cdot \log(k)) = \Theta(N_q \cdot \log(k))$. In practice, however, the array implementation seems to be superior to the hash-based implementation, for two reasons. First, an array is very cache-efficient; a hash table, due to the nonsequential access pattern, could potentially lead to a large number of CPU cache misses. Second, and more important, real-world implementations following the term-at-a-time approach usually do not keep the full set of accumulators, but instead employ a strategy known as *accumulator pruning*.

Accumulator pruning

As mentioned before, one motivation behind term-at-a-time query processing is that the query terms' postings lists may be too large to be completely loaded into memory. Of course, this implies that the accumulator set *acc* is also too large to be kept in memory. Thus, the algorithm in Figure 5.4 is incompatible with its initial motivation. To overcome this inconsistency, we have to revise the algorithm so that it requires only a fixed amount of memory for its accumulator set. That is, we have to impose an upper limit a_{max} on the number of accumulators that may be created.

The two classic accumulator pruning strategies are the QUIT and CONTINUE methods due to Moffat and Zobel (1996). With the QUIT strategy, the search engine — whenever it is done with the current query term — tests whether $|acc| \geq a_{max}$. If this is the case, query processing is terminated immediately, and the current accumulator set represents the final search results. If CONTINUE is used instead, the search engine may continue to process postings lists and to update existing accumulators, but may no longer create any new accumulators.

Unfortunately, since a term's postings list may contain more than a_{max} entries, neither QUIT nor CONTINUE actually enforces a hard limit, and for small values of a_{max} it is in fact quite likely that the limit is exceeded greatly. Our discussion of accumulator pruning is based on more recent work conducted by Lester et al. (2005). Lester's algorithm guarantees that the number of accumulators created stays within a constant factor of a_{max}. In the variant presented here, it is guaranteed that the accumulator limit is never exceeded.

Similar to the MAXSCORE heuristic for document-at-a-time query evaluation, the basic idea behind accumulator pruning is that we are not interested in the full set of matching documents, but only in the top k, for some small k. In contrast to MAXSCORE, however, accumulator pruning strategies are not guaranteed to return the same result set that an exhaustive evaluation would generate. They produce only an approximation; the quality of this approximation depends on the pruning strategy that is used and on the value of a_{max}.

In the algorithm shown in Figure 5.4, the query terms are processed in order of frequency, from least frequent to most frequent. While this is generally beneficial for unpruned term-at-a-time query processing, because fewer accumulators have to be copied from *acc* to *acc'*, it is crucial if accumulator pruning is employed. The main insight is the following: If we are allowed

only a limited number of accumulators, we should spend them on documents that are most likely to make it into the top k. All other things being equal, if query term t has a greater weight than query term t', then a document containing t is more likely to be among the top k than a document containing t'. In the context of BM25 (or any other IDF-based ranking formula), this means that less frequent terms (with shorter postings lists) should be processed first.

In order to define the actual pruning strategy, we have to devise a rule that tells us for a given posting whether it deserves its own accumulator or not. Suppose the search engine, after processing the first $i-1$ query terms, has piled up a set of $a_{current}$ accumulators and is about to start working on t_i. Then we can distinguish three possible cases:

1. $a_{current} + N_{t_i} \leq a_{max}$. In this case we have enough free accumulators for all of t_i's postings, and no pruning is required.
2. $a_{current} = a_{max}$. In this case the accumulator limit has already been reached. None of t_i's postings will be allowed to create new accumulators.
3. $a_{current} < a_{max} < a_{current} + N_{t_i}$. In this case we may not have sufficient accumulator quota to create new accumulators for all of t_i's postings. Pruning may be required.

Cases 1 and 2 are straightforward. For case 3, a possible (simplistic) pruning strategy is to allow new accumulators for the first $a_{max} - a_{current}$ postings whose documents do not yet have accumulators, but not for the remaining ones. The problem with this approach is that it favors documents that appear early on in the collection. This seems inappropriate, as there is no evidence that such documents are more likely to be relevant to the given query (unless documents are sorted according to some query-independent quality measure, such as PageRank).

Ideally, we would like to define a threshold ϑ such that exactly $a_{max} - a_{current}$ postings from t_i's postings list have a score contribution greater than ϑ. Postings whose score exceeds ϑ would then be allowed to create a new accumulator, but lower-scoring postings would not. Taken literally, this approach necessitates a two-pass query processing strategy: In the first pass, score contributions for all of t_i's postings are scored and sorted to obtain the threshold ϑ. In the second pass, all of t_i's postings are scored again, and compared to the threshold. As t_i's postings list may be long, the computational cost of this approach can be substantial. We address this problem in two steps:

1. The necessity to score all of t_i's postings can be eliminated by employing a slightly different threshold value ϑ_{TF} and using it as a proxy for the original threshold ϑ. The new threshold value is compared directly to each posting's TF component, not to its score contribution, thus reducing the overall computational cost.
2. The second pass over t_i's postings can be avoided by using an approximate value of ϑ_{TF} instead of the exact value. The approximate value is computed on the fly and is updated periodically, based on the TF distribution of the postings that have already been processed.

```
   rankBM25_TermAtATimeWithPruning (⟨t_1, ..., t_n⟩, k, a_max, u) ≡
1     sort ⟨t_1, ..., t_n⟩ in increasing order of N_{t_i}
2     acc ← {},  acc' ← {}   // initialize two empty accumulator sets
3     acc[0].docid ← ∞   // end-of-list marker
4     for i = 1 to n do
5        quotaLeft ← a_max − length(acc)   // the remaining accumulator quota
6        if N_{t_i} ≤ quotaLeft then   // Case 1: no pruning required
7           do everything as in Figure 5.4
8        else if quotaLeft = 0 then   // Case 2: no unused accumulators left
9           for j = 1 to length(acc) do
10             acc[j].score ← acc[j].score + log(N/N_{t_i}) · TF_BM25(t_i, acc[j].docid)
11       else   // Case 3: some accumulators left; pruning may be required
12          tfStats[j] ← 0  ∀j   // initialize TF statistics used for pruning
13          ϑ_TF ← 1   // initialize TF threshold for new accumulators
14          postingsSeen ← 0   // the number of postings seen from t_i's postings list
15          inPos ← 0   // current position in acc
16          outPos ← 0   // current position in acc'
17          for each document d in t_i's postings list do
18             while acc[inPos] < d do   // copy accumulators from acc to acc'
19                acc'[outPos++] ← acc[inPos++]
20             if acc[inPos].docid = d then   // term and accumulator coincide
21                acc'[outPos].docid ← d
22                acc'[outPos++].score ← acc[inPos++].score + log(N/N_{t_i}) · TF_BM25(t_i, d)
23             else if quotaLeft > 0 then
24                if f_{t_i,d} ≥ ϑ_TF then   // f_{t_i,d} exceeds the threshold; create a new accumulator
25                   acc'[outPos].docid ← d
26                   acc'[outPos++].score ← log(N/N_{t_i}) · TF_BM25(t_i, d)
27                   quotaLeft ← quotaLeft − 1
28                tfStats[f_{t_i,d}] ← tfStats[f_{t_i,d}] + 1
29             postingsSeen ← postingsSeen + 1
30             if (postingsSeen mod u = 0) then   // recompute ϑ_TF based on tfStats
31                q ← (N_{t_i} − postingsSeen)/postingsSeen
32                ϑ_TF ← argmin_x {x ∈ ℕ | Σ_{j=1}^{x} (tfStats[j] · q) ≥ quotaLeft}
33          copy the remaining accumulators from acc to acc', as in Figure 5.4
34          swap acc and acc'
35    return the top k items of acc   // use a heap to select the top k
```

Figure 5.5 Term-at-a-time query processing with BM25 and accumulator pruning. Input parameters: the query terms $t_1, \ldots, t_n$; the number k of documents to be returned; the accumulator limit a_{max}; the threshold update interval u.

The resulting algorithm is shown in Figure 5.5. In addition to the query terms and the parameter k that specifies the number of search results to return, the algorithm takes two arguments: the accumulator limit a_{max} and the update interval u used for the approximation of ϑ_{TF}. The algorithm uses the array *tfStats* to record the number of postings with a given TF value that,

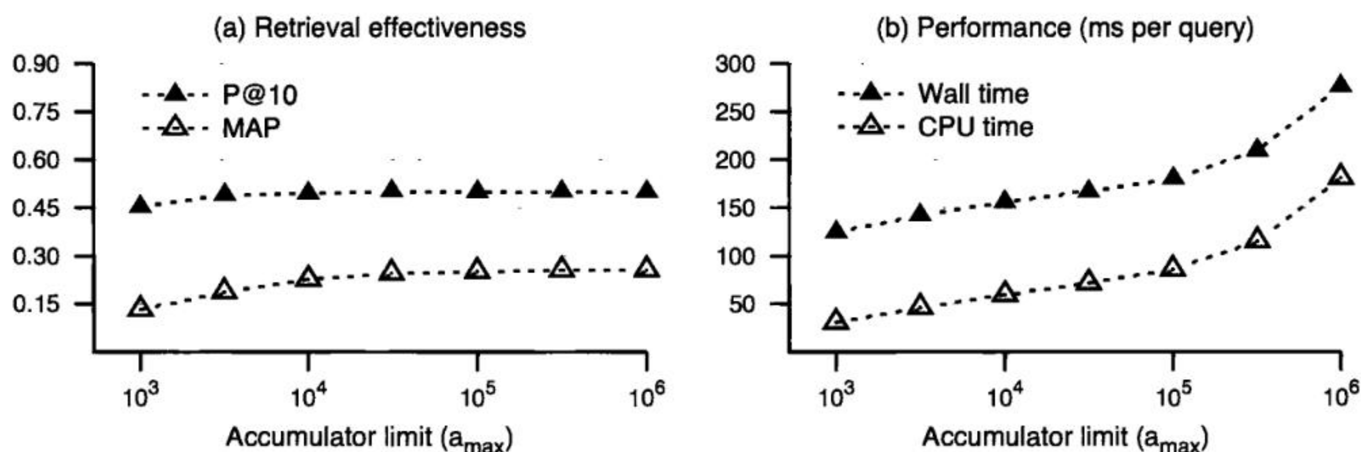


Figure 5.6 Retrieval effectiveness and query processing performance for term-at-a-time query evaluation with accumulator pruning. Data set: 10,000 queries from TREC TB 2006, run against a frequency index for GOV2.

when they were encountered, did not yet have a corresponding accumulator. By extrapolating these statistics to the remaining postings, we can predict how many postings we will see for a given TF value, thus allowing us to compute an estimate for ϑ_{TF}. This computation takes place in lines 31–32 of the algorithm.

The update interval parameter u is used to limit the computational cost of the periodic recomputation of ϑ_{TF}. From our experience, $u = 128$ seems to be a good trade-off between approximation accuracy and computational overhead. As an alternative to a fixed update interval, Lester et al. (2005) suggest exponentially increasing intervals, taking into account the fact that, as we progress, our approximation of the optimal threshold ϑ_{TF} becomes more accurate and requires less frequent corrections.

Regarding the efficiency of the threshold computation in line 32, it is worth pointing out that it is not necessary to have an entry for every possible TF value in the *tfStats* array. Most $f_{t_i,d}$ values are very small (less than 4 or 5), so it makes sense to group larger values together, to avoid maintaining a *tfStats* array in which most entries are 0. For example, changing line 28 of the algorithm to

$$\mathit{tfStats}[\min\{15, f_{t_i,d}\}] \leftarrow \mathit{tfStats}[\min\{15, f_{t_i,d}\}] + 1$$

has rarely any effect on the value of ϑ_{TF}, but allows us to recompute ϑ_{TF} in line 32 very efficiently.

Figure 5.6 shows the retrieval effectiveness and query processing performance of our implementation of the accumulator pruning algorithm in Figure 5.5. Effectiveness was measured by retrieving the top $k = 10{,}000$ documents for short queries generated for the 99 adhoc topics from TREC Terabyte 2004 and 2005. Query processing performance was measured by retrieving the top $k = 10$ documents for each of the 10,000 efficiency topics from TREC Terabyte 2006.

Retrieval effectiveness is quite stable, even for small values of a_{max}. Mean average precision (MAP) starts degrading at $a_{max} \approx 10^5$. Precision at 10 documents (P@10) shows signs of

deterioration only for very aggressive pruning ($a_{max} < 10^4$). The performance of the algorithm, measured in seconds per query, is in the same ballpark as the performance of the document-at-a-time algorithm (with MAXSCORE). For $a_{max} = 10^5$, both methods require approximately the same amount of time and achieve roughly the same level of effectiveness. For larger values of a_{max}, the term-at-a-time method loses to document-at-a-time due to the overhead associated with traversing the whole accumulator array for every query term.

Interestingly, for aggressive pruning ($a_{max} \leq 10^4$), the term-at-a-time algorithm is actually faster than the Boolean AND approach (comparing Figure 5.6 with Table 5.1). This is not entirely unexpected. According to Table 5.1, if the search engine follows a conjunctive query interpretation (Boolean AND), it has to score 28,000 matching documents per query on average. Thus, under certain conditions the conjunctive approach may in fact be less efficient than the disjunctive approach with accumulator pruning. Of course, this comparison is not completely fair, as conjunctive query evaluation is also amenable to accumulator pruning (see Exercise 5.2).

5.1.3 Precomputing Score Contributions

It should have become clear by now that the computation of the final document scores, for example, according to the BM25 formula (Equation 5.5 on page 138), is a major bottleneck of the search engine's query processing routines. Both MAXSCORE and accumulator pruning obtain their performance gains from limiting score computations to documents that are likely to make it into the top k search results.

For scoring algorithms that follow the bag-of-words paradigm (Equation 5.16), it is not necessary to compute the query term's score contributions at query time. Instead, we may precompute each posting's score contribution during index construction and store it in the index, along with the docid, leading to postings of the form (*docid, score*) instead of (*docid, tf*). Precomputing score contributions can dramatically reduce the search engine's CPU cost during query processing. Of course, if TF values are replaced with precomputed score contributions, it will be impossible to make any changes to the scoring functions after the index has been built. In many applications this may not be a problem. But it makes it difficult to experiment with new scoring functions.

A perhaps more serious problem is posed by the space requirements of precomputed scores. To reduce their space requirements (and, in the case of on-disk indices, the disk I/O overhead), inverted indices are usually stored in compressed form. The details of how index compression works are not important here (they are described in Chapter 6), but the basic idea is that small integer values should be stored using a small number of bits. The vast majority of the TF values in an inverted index, for example, are very small (less than 4). Thus, they can be compressed extremely well and require only between 2 and 3 bits per posting in the index. Precomputed

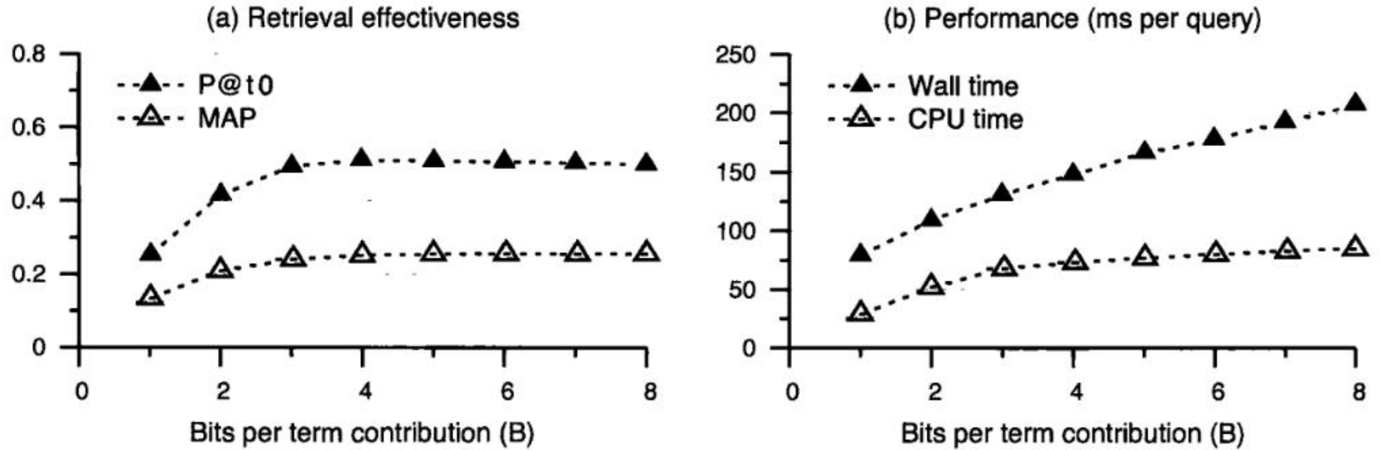


Figure 5.7 Retrieval effectiveness and query processing performance for document-at-a-time query evaluation with precomputed term contributions and MAXSCORE heuristic. Data set: 10,000 queries from TREC TB 2006, run against a frequency index for GOV2.

term scores, on the other hand, are essentially incompressible. If term scores are represented as 32-bit floating point numbers, they will require between 24 and 32 bits in the index.[1]

If we want to speed up the scoring process by using precomputed term contributions, then it is crucial that we discretize the range of possible scores into a predefined set of buckets. Let B denote the number of bits (uncompressed) that we are willing to reserve for each score contribution. Then a possible discretization could be

$$score' = \left\lfloor \frac{score}{score_{max}} \cdot 2^B \right\rfloor, \tag{5.18}$$

where $score_{max}$ is the maximum score contribution allowed by the scoring function. In the case of standard BM25, we have $score_{max} = k_1 + 1 = 2.2$ (ignoring the IDF component of the scoring formula).

Because the discretized $score'$ values are usually not uniformly distributed across all 2^B possible values, they are likely to be amenable to compression, such as Huffman coding (Section 6.2.2), and can be expected to consume slightly less than B bits each.

Figure 5.7 depicts experimental results (retrieval effectiveness and time per query) for a document-at-a-time implementation based on precomputed score contributions. It can be seen that, with as little as 4 bits per precomputed score (uncompressed), we can achieve a retrieval effectiveness that is on par with the original implementation that computes all term scores on the fly. At the same time the average time per query is reduced from 188 ms to 148 ms (-21%).

[1] The IEEE standard for floating point arithmetic divides a 32-bit floating point number into a 24-bit significand and an 8-bit exponent. The significand is virtually incompressible.

Increasing the number of bits per score makes the search engine slower, up to a point where the revised implementation, using precomputed scores, requires more time per query than the original implementation (e.g., 207 ms per query for $B = 8$, up from 188 ms). Note, however, that the slowdown stems almost exclusively from the increased disk I/O overhead, as the average CPU time per query remains approximately constant. Thus, the effect will not be observed if the index is kept in memory instead of on disk.

5.1.4 Impact Ordering

In the inverted index data structure described in Chapter 4, postings are stored in the index in the order in which they appear in the collection. This way of organizing the postings lists has the advantage that the index is easy to build and that query operations (such as intersecting postings lists) can be carried out relatively efficiently. However, the original document order is not necessarily the best ordering for the postings in a given postings list. In the term-at-a-time algorithm with accumulator pruning (Figure 5.5), for instance, we had to resort to a few tricks to efficiently obtain an approximation of the top-scoring postings in t_i's postings list. If the postings in that list had been sorted by their score contributions, this would have been much easier; we could have selected the first few postings from the list and ignored the rest.

An index in which the postimgs in each list are sorted according to their respective score contributions is called an *impact-ordered* index. Impact-ordered indices usually contain precomputed scores instead of TF values, as the ordering of the postings already implies a certain scoring function.

When implemented naïvely, impact ordering can have a devastating effect on the complexity of basic index access functions. For instance, the **next** method from Chapter 2 no longer has logarithmic, but linear complexity! To avoid this, postings in an impact-ordered index are usually not ordered by their exact score contribution, but — similar to what we did in the previous section — according to a quantized impact value. As before, we can group impact values into 2^B discrete buckets, where B is typically between 3 and 5. The postings in a given term's postings list are then sorted by their discretized impact value. Within each bucket, however, postings are sorted according to docid, as in the standard index organization from Chapter 4.

Following this hybrid approach, the computational overhead of random access operations, compared to a strictly document-ordered index, becomes tolerable. And the advantages of having quick access to a term's top postings are quite obvious, especially within a term-at-a-time query processing framework.

5.1.5 Static Index Pruning

The accumulator pruning technique from Section 5.1.2, with or without impact ordering, achieves its performance gains by ignoring a large portion of the query terms' postings. For instance, with an accumulator limit $a_{max} = 1{,}000$, the search engine never scores more than

1,000 documents. Unfortunately, even though many postings do not contribute anything to the scoring process, most of them still need to be read from the index and decompressed in order for the search engine to be able to access the "interesting" postings that do make a contribution.

If we want to squeeze the last bit of query processing performance out of our search engine, we can try to predict — at indexing time — which postings are likely to be used during query processing and which are not. Based on this prediction, we can remove from the index all postings which we don't think will play an important role in the search process. This strategy is referred to as *static index pruning*. It can yield substantial performance improvements due to the shorter postings lists in the index.

Static index pruning has several obvious limitations. First, when processing queries from a pruned index, we can no longer guarantee that the search engine returns the top-scoring documents for a given query. Second, it breaks down if we allow the users to define structural query constraints or to issue phrase queries. For instance, if a user searches for the phrase "to be or not to be" in the Shakespeare collection, and the index pruning algorithm has decided that the occurrence of "not" in act III, scene 2 of Shakespeare's *Hamlet* is unimportant, then the user will not be able to find the scene that she is looking for. Nonetheless, despite being inappropriate for certain types of queries, index pruning has proven to be quite successful for traditional bag-of-words query processing algorithms.

Index pruning algorithms usually come in one of two flavors:

- In *term-centric* index pruning, each term in the index is treated independently. From each term's postings list, the pruning algorithm selects the most important postings, according to some predefined criteria, and discards the rest.
- A *document-centric* pruning algorithm, in contrast, looks at individual documents. Given a document, the algorithm tries to predict which query terms are most important and most representative of that document. If a term is considered important, a posting is added to the index; otherwise, the term is discarded and treated as if it did not appear in the document at all.

Term-centric pruning

Term-centric index pruning was first studied by Carmel et al. (2001). In their paper they examine several different pruning algorithms that are, however, all relatively similar. Here we limit ourselves to their top-(K,ε) term-centric pruning algorithm.

Consider a term t with associated postings list L_t, and an unknown query q that contains the term t. Everything else being equal, the probability that a given posting P in L_t corresponds to one of the top k documents for the query q is monotonic in P's score contribution. Therefore, the term-centric pruning algorithm operates by sorting all postings in L_t according to their score contribution for the chosen scoring function (e.g., BM25). It selects the top K_t postings for inclusion in the index, discarding all other elements of L_t.

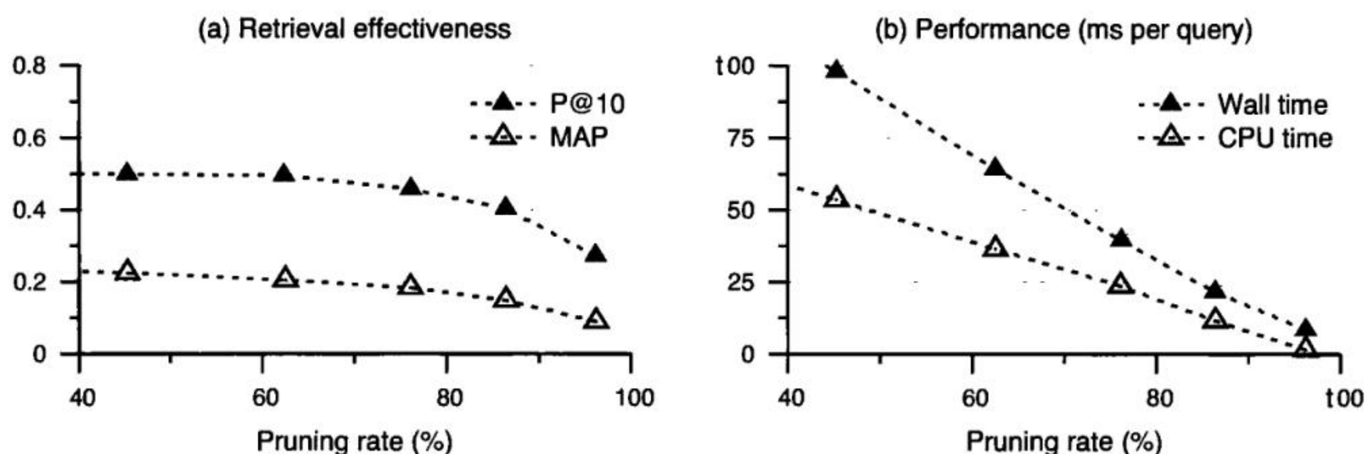


Figure 5.8 Term-centric index pruning with $K = 1{,}000$ and ε between 0.5 and 1. Data set for efficiency evaluation: 10,000 queries from TREC TB 2006. Data set for effectiveness evaluation: TREC topics 701–800.

There are various ways in which the cutoff parameter K_t can be chosen. One possibility is to use the same value K_t for all terms in the index. Alternatively, an index-wide score contribution threshold ϑ may be chosen, such that all postings whose score contribution exceeds ϑ are included in the index, while all other postings are discarded. A third variant is to guarantee each term at least K postings in the index. If a term t appears in more than K documents, its postings limit K_t depends on the distribution of score contributions in its postings list. This is the top-(K,ε) pruning method:

- Choose two parameters $K \in \mathbb{N}$ and $\varepsilon \in [0, 1]$.
- If a term t appears in fewer than K documents, store all of t's postings in the index.
- If a term t appears in more than K documents, compute $\vartheta_t = score(L_t[K]) \cdot \varepsilon$, where $score(L_t[K])$ is the score contribution of t's Kth-highest-scoring posting. Discard all postings with a score contribution below ϑ_t. Include the remaining postings in the index.

In general, it is unclear how K and ε should be chosen. However, once one parameter has been fixed, the other one can be varied freely until the desired index size, performance, or search result quality is reached.

In our experiments (Figure 5.8) we arbitrarily set $K = 1000$ and measured both retrieval effectiveness and query performance for various values of ε between 0.5 and 1. As can be seen from the figure, for $\varepsilon = 0.5$ (pruning rate: ≈50%) there is no noticeable difference in result quality between the pruned and the unpruned index (P@10 = 0.500, down from 0.503; MAP = 0.238, down from 0.260). The average time per query, however, decreases from 188 ms to 118 ms (-37%). For more aggressive pruning parameters, the performance gains are even more pronounced. However, if more than 70% of all postings are removed from the index, the quality of the search results begins to suffer markedly.

Document-centric pruning

Document-centric index pruning is motivated by document-based pseudo-relevance feedback methods (see Section 8.6). Pseudo-relevance feedback is a two-pass query processing strategy in which, in a first pass, the search engine identifies the set of k' top-scoring documents for the given query. It then assumes that these k' documents are all relevant to the query (*pseudo-relevance*) and selects a set of terms that it considers most representative of these documents. The selection is usually based on a statistical analysis of term distributions. The selected terms are then added to the original query (usually with a reduced weight, so that they do not outweigh the original query terms), and the augmented query is used for scoring in a second retrieval pass. It turns out that the set of terms selected by pseudo-relevance feedback often contains the original query terms. Hence, it is possible to run a query-independent pseudo-relevance feedback algorithms on individual documents — at indexing time! — to select a set of terms for which the given document is likely to rank highly.

Inspired by a pseudo-relevance feedback mechanism studied by Carpineto et al. (2001), Büttcher and Clarke (2006) propose the following document-centric pruning algorithm:

- Select a pruning parameter $\lambda \in (0, 1]$.
- For each document d, sort all n unique terms in d according to the following function:

$$score(t,d) \;=\; p_d(t) \cdot \log\left(\frac{p_d(t)}{p_{\mathcal{C}}(t)}\right), \tag{5.19}$$

  where $p_d(t) = f_{t,d}/l_d$ is t's probability of occurrence according to the unigram language model of the document d, and $p_{\mathcal{C}}(t) = l_t/l_{\mathcal{C}}$ is the term's probability of occurrence according to the unigram language model of the text collection $\mathcal{C}$. Select the top $\lceil \lambda \cdot n \rceil$ terms for inclusion in the index; discard the rest.

If you are familiar with the concepts of information theory, Equation 5.19 might remind you of the *Kullback-Leibler divergence* (*KL divergence*) between two probability distributions. Given two discrete probability distributions f and g, their KL divergence is defined as

$$\sum_x f(x) \cdot \log\left(\frac{f(x)}{g(x)}\right). \tag{5.20}$$

KL divergence is often used to measure how different two probability distributions are. It plays an important role in various areas of information retrieval. In Section 9.4 it is used to compute document scores in a language modeling framework for ranked retrieval.

In the context of index pruning, we can use KL divergence to quantify how different a given document is from the rest of the collection. The pruning criterion from Equation 5.19 can then be viewed as selecting the terms that make the greatest contribution to the document's KL divergence and thus are, in some sense, most representative of what makes the document

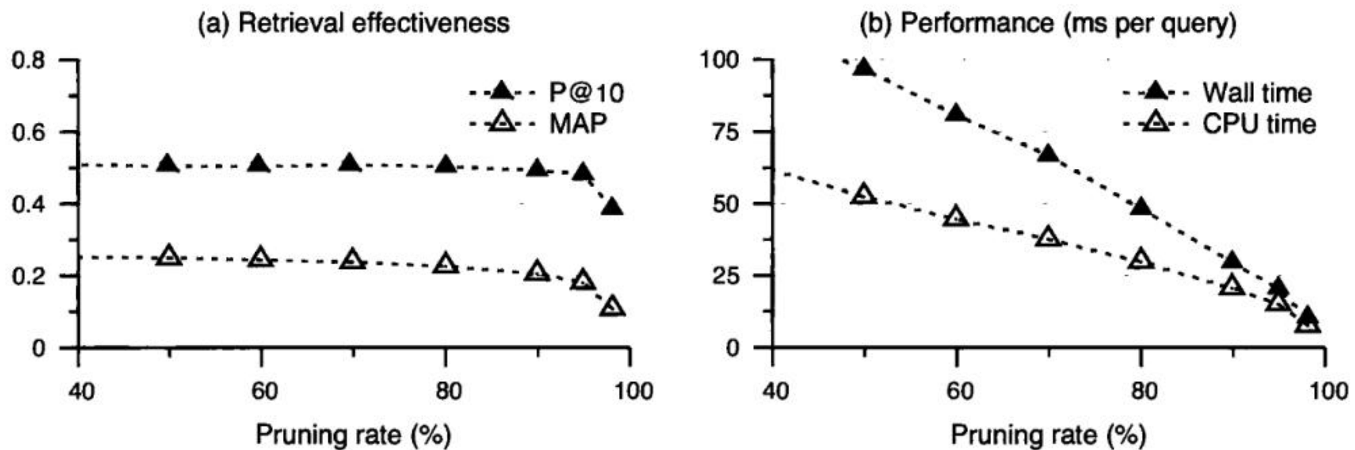


Figure 5.9 Document-centric index pruning with λ between 0.05 and 0.6. Data set for efficiency evaluation: 10,000 queries from TREC TB 2006. Data set for effectiveness evaluation: TREC topics 701–800.

unique. Note that this pruning criterion is independent of the scoring function employed by the search engine.

Figure 5.9 shows experimental results obtained for document-centric pruning. You can see that moderate levels of pruning have virtually no impact on retrieval effectiveness. For instance, after removing 70% of the postings from the index ($\lambda = 0.3$), MAP is still at 0.238 (down from 0.260). P@10 is almost unaffected and is even slightly higher than for an unpruned index (0.508 vs. 0.503). Only if we prune very aggressively ($\lambda < 0.1$), we will start to see an impact on early precision (P@10).

If you compare Figures 5.8 and 5.9, you will see that, at any given pruning rate, the term-centric method leads to lower response times than the document-centric approach. This can be explained by the fact that term-centric pruning selects postings for all terms in the index. Many of those terms are content generation artifacts (e.g., document time stamps) that never show up in search queries. The document-centric method detects that these terms are not important within their respective documents (because they usually appear only a single time in the document). It excludes them from the index, which increases the average length of the postings lists for the remaining terms. Therefore, at the same pruning level, the document-centric approach leads to more work for the search engine.

Figure 5.10 shows a head-to-head comparison between term-centric and document-centric pruning. For moderate performance levels (> 60 ms per query), there is not much of a difference between the two methods; both achieve almost the same precision as the original unpruned index. As we push for higher performance, however, the difference becomes quite visible. For example, at a fixed precision level of P@10=0.45, document-centric pruning is about twice as fast as the term-centric variant.

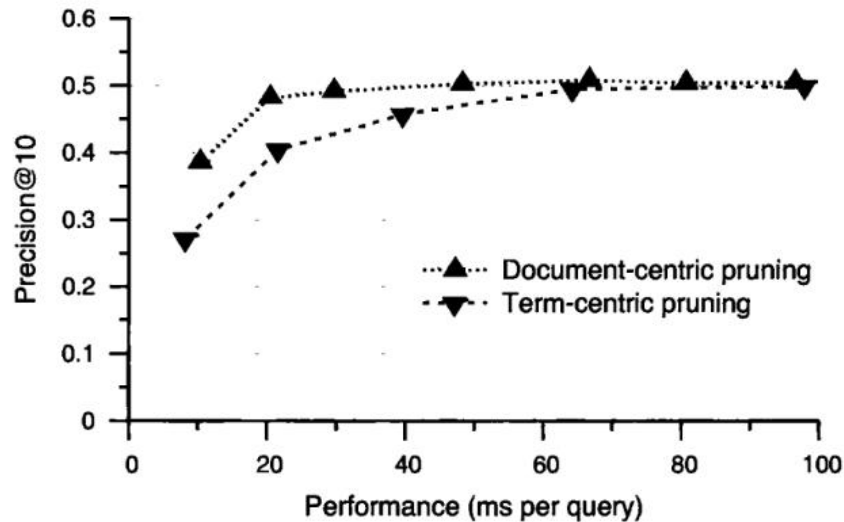


Figure 5.10 Efficiency-vs.-effectiveness trade-offs for document-centric and term-centric static index pruning.

Correctness guarantees

Although the static index pruning methods discussed above have interesting properties and allow the operator of a search engine to make efficiency-versus-effectiveness trade-offs by changing the parameters of the pruning algorithm, they are often not very attractive in real-world settings. In the Web search market, for instance, competition is fierce, and even small reductions in result quality represent a competitive disadvantage that may lead to a reduced market share. Performance improvements are welcome, but only as long as they don't have a negative impact on retrieval effectiveness.

Ntoulas and Cho (2007) present a method that can be used to decide whether the search results obtained from a pruned index are identical with the search results an unpruned index would have produced. Their work is based on the assumption that the ranking function employed by the search engine follows a bag-of-words scoring model (see Equation 5.16 on page 145).

For simplicity, our presentation of Ntoulas's method is based on a modified version of Equation 5.16 that lacks the quality component:

$$\text{score}(q, d) \;=\; \sum_{t \in q} \text{score}(t, d). \tag{5.21}$$

Now suppose that the search engine uses a scoring function that is of the form shown above, and that it processes queries from a pruned index, with some postings missing. Further suppose that the term-centric top-(K,ε) pruning mechanism from page 154 was used to prune the index. Then, for each term t in the index, there is a threshold value ϑ_t, such that all of t's postings

```
    rankBM25_DocumentAtATimeWithCorrectnessGuarantee (⟨t_1, ..., t_n⟩, k) ≡
1       m ← 0   // m is the total number of matching documents
2       d ← min_{1≤i≤n} {nextDoc(t_i, −∞)}
3       while d < ∞ do
4           results[m].docid ← d
5           results[m].score ← 0
6           results[m].numHits ← 0
7           for i ← 1 to n do
8               if nextDoc(t_i, d − 1) = d then
9                   results[m].score ← results[m].score + log(N/N_{t_i}) · TF_BM25(t_i, d)
10                  results[m].numHits ← results[m].numHits + 1
11              else
12                  results[m].score ← results[m].score + ϑ_{t_i}
13          m ← m + 1
14          d ← min_{1≤i≤n} {nextDoc(t_i, d)}
15      sort results[0..(m − 1)] in decreasing order of score
16      if results[i].numHits = n for 0 ≤ i < k then
17          return (results[0..(k − 1)], true)   // results are guaranteed to be correct
18      else
19          return (results[0..(k − 1)], false)   // results may be incorrect
```

Figure 5.11 Document-at-a-time query processing based on a pruned index. Depending on whether the correctness of the top k search results can be guaranteed or not, the function returns either *true* or *false*.

with a term contribution in excess of ϑ_t are in the index, while all postings with a smaller contribution have been dropped.

We may use this fact to determine whether the results produced by the pruned index are the same as the results that an unpruned index would have produced or whether they might be different. Consider a three-word query $q = \langle t_1, t_2, t_3 \rangle$ and a document d that, according to the pruned index, contains t_1 and t_2, but not t_3. From the index alone, it is impossible to tell whether d actually contains t_3 or not, as the posting may have been pruned. However, we do know that, if d contains t_3, then score(t_3, d) cannot be larger than ϑ_{t_3}. This gives us an upper bound for d's final score:

$$\text{score}(q, d) \;\leq\; \text{score}(t_1, d) + \text{score}(t_2, d) + \vartheta_{t_3}. \tag{5.22}$$

We can take into account the incomplete nature of the pruned index by ranking the document d not according to its computed score but according to this upper bound. The resulting algorithm, a revised version of the algorithm in Figure 5.1, is shown in Figure 5.11. In the algorithm, when a document does not have a hit for query term t, the pruning cutoff ϑ_t is used in place of the actual score contribution. At the end of the computation, if we have complete information for each of the top k results ($results[i].numHits = n$ in line 16), we know that the ranking is correct.

The algorithm in Figure 5.11 can be used as part of a two-tiered query processing architecture in which the first tier consists of a pruned index and the second tier contains the original, unpruned index. Queries are first sent to the pruned index. If we can guarantee the correctness of the results produced by the pruned index, we are done. Otherwise, we process the query again, this time using the unpruned index. Depending on the relative performance of the pruned index compared to the full index, and on the fraction of queries that can be answered without consulting the full index, this approach has the potential to reduce the average amount of work done per query without sacrificing the quality of the search results. However, note that the algorithm in Figure 5.11 cannot be used in conjunction with document-centric pruning, because the latter does not provide an upper bound ϑ_t for the score contribution of pruned term occurrences. Thus, although document-centric pruning often does not reduce the quality of the search results, the index does not contain enough information for us to prove it.

5.2 Lightweight Structure

At the start of Chapter 2 we presented a simple abstract data type (ADT) for inverted indices. We demonstrated how this ADT could be used for phrase searching and for computing simple relationships involving structural elements, such as identifying all speeches spoken by a particular character in Shakespeare's plays (page 45). In that example we explicitly used the methods of the ADT to find occurrences of the character's name, and then the speeches containing them. We now extend and generalize this approach through the presentation of a *region algebra*.

Region algebras provide operators and functions for combining and manipulating text intervals (or *regions*) in support of lightweight structure. They represent an intermediate point between basic document retrieval and the complexity of full XML retrieval (see Chapter 16). They also provide a method for unifying many of the *advanced search* features supported by search engines and digital libraries. A number of region algebras have been described in the literature, dating back to the PAT region algebra, created for the *New Oxford English Dictionary* project in the early 1980s (Gonnet, 1987; Salminen and Tompa, 1994). The particular region algebra we present in this chapter is representative of the group, but is relatively simple and builds directly on the techniques introduced in Chapter 2 (Clarke et al., 1995a, 1995b; Clarke and Cormack, 2000).

5.2.1 Generalized Concordance Lists

Broadly speaking, region algebras work with sets of text intervals. Each interval may be expressed as a pair $[u, v]$, where u indicates the start of the interval and v indicates its end. Our particular region algebra places a simple but important requirement on the sets of intervals it manipulates: *No interval in the set may have another interval from the set nested within it.* We refer to a set of intervals with this property as a *generalized concordance list*, or *GC-list*.

The GC-list takes its name from a *concordance*, an alphabetical listing of the words in a document along with the context in which they appear. Prior to the advent of computer-based retrieval, paper-based concordances provided a tool for searching major works such as Shakespeare's. Essentially, they were an early form of an inverted index.

We use the notation $[u, v] \sqsubset [u', v']$ to indicate that $[u, v]$ is nested in $[u', v']$. Similarly, the notation $[u, v] \not\sqsubset [u', v']$ indicates that $[u, v]$ is not nested in $[u', v']$. If $[u, v]$ and $[u', v']$ are both part of a set manipulated by our region algebra, then $[u, v] \not\sqsubset [u', v']$ — either $u < u'$ and $v < v'$, or $u > u'$ and $v > v'$. For example, the set of intervals

$$S = \{[1, 10], [5, 9], [8, 12], [15, 20]\} \tag{5.23}$$

is not a GC-list because $[5, 9] \sqsubset [1, 10]$. We could reduce this set to a GC-list by, for example, removing $[1, 10]$ to produce

$$\{[5, 9], [8, 12], [15, 20]\}. \tag{5.24}$$

Note that $[5, 9]$ *overlaps* $[8, 12]$. Unlike nesting intervals, overlapping intervals are acceptable. Indeed, overlaps are essential to the correct operation of the region algebra. Note that we could also reduce S to a GC-list by removing the interval $[5, 9]$. For reasons that will become apparent later in this section, eliminating the larger interval is preferred.

We formalize the reduction of a set of intervals to a GC-list as follows: Let S be a set of intervals. We define the function $\mathcal{G}(\mathrm{S})$ as

$$\mathcal{G}(S) = \{a \mid a \in S \text{ and } \nexists\, b \in S \text{ such that } b \sqsubset a\} \tag{5.25}$$

Given a set of text intervals, this function eliminates those that have other members of the set nested within them, thus reducing the set to a GC-list. Therefore, a set S is a GC-list if and only if $\mathcal{G}(S) = S$ and $\mathcal{G}(S) = \mathcal{G}(\mathcal{G}(S))$. GC-lists arise naturally from our implementation of the region algebra. While our implementation never explicitly reduces a set of intervals to a GC-list, the function helps us to provide a concise explanation for the behavior of the algebra.

GC-lists possess a number of important properties. Since intervals cannot nest, no two intervals in a GC-list may start at the same position. Thus, given two intervals $[u, v]$ and $[u', v']$, either $u < u'$ or $u > u'$. Similarly, no two intervals may end at the same position; if $u < u'$, then $v < v'$. Thus, ordering the intervals in a GC-list by either their start positions or their end positions produces the same ordering. Finally, because each position can form the start position for no more than one interval, the size of a GC-list is bounded by the total length of the collection.

A schema-independent postings list may be viewed as a GC-list by treating each entry as an interval of length 1, starting and ending at the same position. Thus, the postings list for "first" in Figure 2.1 on page 34 may be viewed as the GC-list

$$\text{“first”} \;=\; \{[2205, 2205], [2268, 2268], ..., [1271487, 1271487]\}. \tag{5.26}$$

Table 5.2 Definitions for the binary operators in the region algebra.

Containment Operators
Contained In:
$A \lhd B = \{a \mid a \in A \text{ and } \exists\, b \in B \text{ such that } a \sqsubset b\}$
Containing:
$A \rhd B = \{a \mid a \in A \text{ and } \exists\, b \in B \text{ such that } b \sqsubset a\}$
Not Contained In:
$A \ntriangleleft B = \{a \mid a \in A \text{ and } \nexists\, b \in B \text{ such that } a \sqsubset b\}$
Not Containing:
$A \ntriangleright B = \{a \mid a \in A \text{ and } \nexists\, b \in B \text{ such that } b \sqsubset a\}$
Combination Operators
Both Of:
$A \bigtriangleup B = \mathcal{G}(\{c \mid \exists\, a \in A \text{ such that } a \sqsubset c \text{ and } \exists\, b \in B \text{ such that } b \sqsubset c\})$
One Of:
$A \bigtriangledown B = \mathcal{G}(\{c \mid \exists\, a \in A \text{ such that } a \sqsubset c \text{ or } \exists\, b \in B \text{ such that } b \sqsubset c\})$
Ordering Operator
Before:
$A \ldots B = \mathcal{G}(\{c \mid \exists\, [u,v] \in A \text{ and } \exists\, [u',v'] \in B \text{ where } v < u' \text{ and } [u,v'] \sqsubset c\})$

The answer to a phrase query may also be viewed as a GC-list, with the start and end positions of each phrase forming an interval in the GC-list. For example, the GC-list corresponding to the phrase "first witch" over the Shakespeare collection is

$$\text{“first witch”} \;=\; \{[745406, 745407], [745466, 745467], [745501, 745502], \ldots\}. \tag{5.27}$$

5.2.2 Operators

Our region algebra has seven binary operators, presented in Table 5.2. Each operator is defined over GC-lists and evaluates to a GC-list. The operators fall into three classes: containment, combination, and ordering.

The *containment operators* select the intervals in a GC-list that are contained in, not contained in, contain, or do not contain the intervals in a second GC-list. The containment operators are used to formulate queries that refer to the hierarchical characteristics of structural elements in the collection. The expression on the right-hand side of a containment operator acts as a filter to restrict the expression on the left-hand side — the result of the operation is a subset of the GC-list on the left-hand side.

The two *combination operators* are similar to the standard Boolean operators AND and OR. The "both of" operator is similar to AND: Each interval in the result contains an interval from both operands. The $\mathcal{G}()$ function is applied to ensure the result is a GC-list. The "one of" operator merges two GC-lists: Each interval in the result is an interval from one of the operands.

The *ordering operator* generalizes concatenation. Each interval in the result starts with an interval from the first operand and ends with an interval from the second operand. The interval from the first operand will end before the starting position of the interval from the second operand. The ordering operator may be used, for example, to connect tags that delineate structural elements, producing GC-lists in which each interval corresponds to one occurrence of the structural element. Examples showing the use of all seven binary operators are given in the next section.

In addition to the binary operators, there are two unary *projection operators*, π_1 and π_2. If A is a GC-list, we define

$$\pi_1(A) = \{[u,u] \mid \exists v \text{ with } [u,v] \in A\}, \tag{5.28}$$

$$\pi_2(A) = \{[v,v] \mid \exists u \text{ with } [u,v] \in A\}. \tag{5.29}$$

For example,

$$\pi_1(\{[5,9],[8,12],[15,20]\}) = \{[5,5],[8,8],[15,15]\}, \tag{5.30}$$

$$\pi_2(\{[5,9],[8,12],[15,20]\}) = \{[9,9],[12,12],[20,20]\}. \tag{5.31}$$

We also define GC-lists that include all intervals of a fixed size as

$$[i] = \{[u,v] \mid v-u+1=i\}. \tag{5.32}$$

For example,

$$[10] = \{...,[101,110],[102,111],[103,112],...\}. \tag{5.33}$$

5.2.3 Examples

The region algebra may be used to solve the queries given on page 45 of Chapter 2, as follows:

1. *Lines spoken by any witch.*

   ("⟨LINE⟩" ... "⟨/LINE⟩")
   ◁ (("⟨SPEECH⟩" ... "⟨/SPEECH⟩")
   ▷ (("⟨SPEAKER⟩" ... "⟨/SPEAKER⟩") ▷ "witch"))

   Brackets group the expressions, indicating the order in which the operations are to be applied. The query first locates speakers that are witches. The lines within their speeches are then extracted. All intermediate results, as well as the final result, are GC-lists.

2. *The speaker who says, "To be or not to be".*

$$
\begin{array}{l}
(\ \text{“}\langle\text{SPEAKER}\rangle\text{”}\ \ ...\ \ \text{“}\langle/\text{SPEAKER}\rangle\text{”}\ )\\
\lhd\ (\ (\ \text{“}\langle\text{SPEECH}\rangle\text{”}\ \ ...\ \ \text{“}\langle/\text{SPEECH}\rangle\text{”}\ )\\
\qquad \rhd\ (\ (\ \text{“}\langle\text{LINE}\rangle\text{”}\ \ ...\ \ \text{“}\langle/\text{LINE}\rangle\text{”}\ )\ \rhd\ \text{“to be or not to be”}\ )\ )
\end{array}
$$

Lines containing the quote are located, the corresponding speeches are identified, and their speakers are extracted. Since this phrase appears only once in Shakespeare's plays, the resulting GC-list contains only a single interval.

3. *Titles of plays mentioning witches and thunder.*

$$
\begin{array}{l}
(\ \text{“}\langle\text{TITLE}\rangle\text{”}\ \ ...\ \ \text{“}\langle/\text{TITLE}\rangle\text{”}\ )\\
\lhd\ (\ (\ \text{“}\langle\text{PLAY}\rangle\text{”}\ \ ...\ \ \text{“}\langle/\text{PLAY}\rangle\text{”}\ )\ \rhd\ (\ \text{“witch”}\ \ \triangle\ \text{“thunder”}\ )\ )
\end{array}
$$

The query first identifies fragments of text that include both "witch" and "thunder", expressing the result as a GC-list. Titles are then extracted from plays containing these fragments.

In these examples we make reasonable assumptions about the structure of the XML used to encode Shakespeare's plays: Elements such as titles, scenes, plays, and lines are surrounded by appropriate tags; all plays contain a title; all speeches contain a speaker and one or more lines. In addition, we assume that structural elements do not nest (i.e., speeches don't contain other speeches).

A unique property of the region algebra is its ability to assign a meaning to a Boolean query such as "witch" $\triangle$ "thunder", without reference to an explicit universe such as plays or lines, since its solution be may expressed as a GC-list. Over Shakespeare's plays,

$$\text{“witch”}\ \triangle\ \text{“thunder”}\ =\ \{[31463, 36898], [36898, 119010], [125483, 137236], ...\}. \qquad (5.34)$$

Intervals in this GC-list overlap, and some intervals extend across multiple plays. Since no interval may have another nested within it, each interval starts with either the term "witch" or the term "thunder", and ends with the other term. Locating plays, scenes, lines, or structural elements that satisfy the Boolean expression is a matter of filtering one GC-list against another.

The combination operators are sufficient for Boolean expression requiring only AND and OR. For example, the query

$$(\ \text{“witch”}\ \triangledown\ \text{“king”}\ )\ \triangle\ (\ \text{“thunder”}\ \triangledown\ \text{“dagger”}\ )$$

identifies fragments of text that contain either "witch" or "king" and either "thunder" or "dagger". Intervals in the resulting GC-list do not have intervals contained within them that also satisfy the Boolean expression. Any larger interval that satisfies the Boolean expression will have an interval from the GC-list contained within it. In our definition of $\mathcal{G}(S)$ in Equation 5.25, we chose to remove the larger of two nesting intervals rather than the smaller. The reasoning

underlying this choice may now be more apparent. At least in the case of Boolean expressions, nothing is gained by explicitly computing these larger intervals.

Boolean NOT requires an explicit universe. For example, a query for plays not mentioning witches or thunder could be expressed as

$$(\text{“}\langle\text{PLAY}\rangle\text{”} \;\ldots\; \text{“}\langle\text{/PLAY}\rangle\text{”}) \not\triangleright (\text{“witch”} \;\triangledown\; \text{“thunder”}).$$

Our region algebra may be used as one method for implementing the advanced search features supported by many search engines and digital libraries. It is fairly common for these IR systems to support Boolean queries that are restricted to specific fields, such as titles, authors, or abstracts. These queries naturally map into the facilities of the region algebra. Many Web search engines support a feature that allows a search to be restricted to a particular site. Including the query term “site:uwaterloo.ca” restricts the search to pages at the University of Waterloo. Assuming appropriate tagging and indexing, this query might be translated into the expression

$$(\text{“}\langle\text{PAGE}\rangle\text{”} \;\ldots\; \text{“}\langle\text{/PAGE}\rangle\text{”}) \triangleright ((\text{“}\langle\text{SITE}\rangle\text{”} \;\ldots\; \text{“}\langle\text{/SITE}\rangle\text{”}) \triangleright \text{“uwaterloo.ca”}).$$

5.2.4 Implementation

The implementation of our region algebra generalizes the algorithms for phrase searching, proximity ranking, and Boolean queries presented in Chapter 2. As we demonstrated earlier, the start positions and the end positions of each interval order the elements of a GC-list in exactly the same way. We exploit this ordering to develop a framework for efficiently implementing the region algebra. The approach involves indexing into GC-lists using an ADT similar to the one we defined for inverted indices on page 33. The ordering of intervals in a GC-list is used as the basis for defining this ADT. Given a GC-list and a position in the collection, we index into the GC-list to find the interval that is in some sense “closest to” that position. We begin with an example and follow it with a formal exposition of the framework.

Consider evaluating the expression $A \ldots B$ (see Figure 5.12). An interval from the resultant GC-list starts with an interval from A and ends with an interval from B. Suppose $[u, v]$ is the first interval in A. If $[u', v']$ is the first interval from B with $u' > v$, then v' must be the end of the first interval of $A \ldots B$. We index into B to find the first interval with $u' > v$. The last interval from A that ends before u' starts the first interval of $A \ldots B$. We index into A to find the last interval $[u'', v'']$ where $v'' < u'$. The interval $[u'', v']$ is then the first solution to $A \ldots B$. Indexing first into A, then into B, and again into A, gives us the first interval in $A \ldots B$ in three steps. The next solution to $A \ldots B$ begins after u''. We index into A to produce the first interval after u''. This procedure of successively indexing into A and B can be continued to find the remaining intervals in $A \ldots B$.

The implementation framework consists of four methods that allow indexing into GC-lists in various ways. Each of the methods represents a variation on the notion of “closest interval”

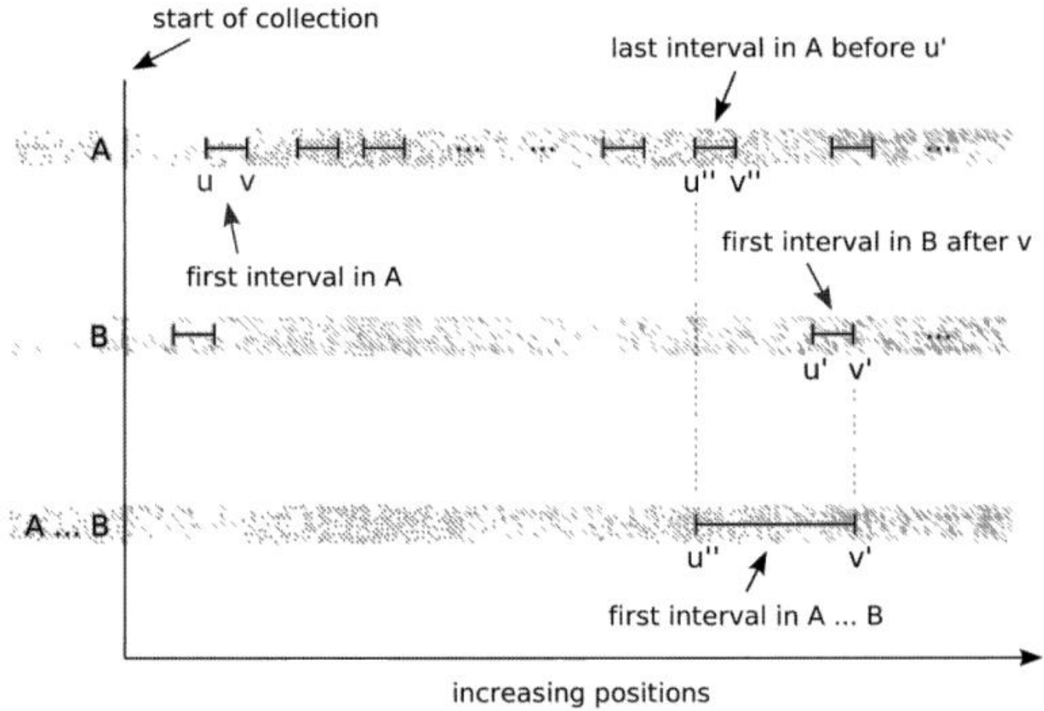


Figure 5.12 Evaluating the GCL expression A ... B.

in a GC-list to a specified position in the collection. We implement the four methods for each operator in the region algebra, using the methods of its operands.

- The method $\tau(S, k)$ returns the first interval in the GC-list S starting at or after the position k:

$$\tau(S,k) = \begin{cases} [u,v] & \text{if } \exists\, [u,v] \in S \text{ such that } k \leq u \\ & \quad\text{and } \nexists\, [u',v'] \in S \text{ such that } k \leq u' < u \\ [\infty,\infty] & \text{if } \nexists\, [u,v] \in S \text{ such that } k \leq u \end{cases} \quad (5.35)$$

- The method $\rho(S, k)$ returns the first interval in S ending at or after the position k:

$$\rho(S,k) = \begin{cases} [u,v] & \text{if } \exists\, [u,v] \in S \text{ such that } k \leq v \\ & \quad\text{and } \nexists\, [u',v'] \in S \text{ such that } k \leq v' < v \\ [\infty,\infty] & \text{if } \nexists\, [u,v] \in S \text{ such that } k \leq v \end{cases} \quad (5.36)$$

```
   τ(t, k) ≡                          τ'(t, k) ≡
1    if k = ∞ then                 8    if k = ∞ then
2      u ← ∞                       9      v ← ∞
3    else if k = −∞ then           10   else if k = −∞ then
4      u ← −∞                      11     v ← −∞
5    else                          12   else
6      u ← next(t, k − 1)          13     v ← prev(t, k + 1)
7    return [u, u]                 14   return [v, v]
```

Figure 5.13 Pseudo-code for $\tau(t,k)$ and $\tau'(t,k)$, where t is a term.

- The method $\tau'(S,k)$ is the converse of τ. It returns the last interval in S ending at or before the position k:

$$\tau'(S,k) = \begin{cases} [u,v] & \text{if } \exists\ [u,v] \in S \text{ such that } k \geq v \\ & \quad \text{and } \nexists\ [u',v'] \in S \text{ such that } k \geq v' > v \\ [-\infty,-\infty] & \text{if } \nexists\ [u,v] \in S \text{ such that } k \geq v \end{cases} \tag{5.37}$$

- The method $\rho'(S,k)$ is the converse of ρ. It returns the last interval in S starting at or before the position k:

$$\rho'(S,k) = \begin{cases} [u,v] & \text{if } \exists\ [u,v] \in S \text{ such that } k \geq u \\ & \quad \text{and } \nexists\ [u',v'] \in S \text{ such that } k \geq u' > u \\ [-\infty,-\infty] & \text{if } \nexists\ [u,v] \in S \text{ such that } k \geq u \end{cases} \tag{5.38}$$

For example, if $S = \{[5,9],[8,12],[15,20]\}$ and $k = 10$, we have:

$$\begin{aligned} \tau(\{[5,9],[8,12],[15,20]\},10) &= [15,20] \\ \rho(\{[5,9],[8,12],[15,20]\},10) &= [8,12] \\ \tau'(\{[5,9],[8,12],[15,20]\},10) &= [5,9] \\ \rho'(\{[5,9],[8,12],[15,20]\},10) &= [8,12] \end{aligned}$$

Just as they do for our inverted index ADT, the symbols ∞ and $-\infty$ act as end-of-file and beginning-of-file markers.

The methods $\tau(S,k)$, $\rho(S,k)$, $\tau'(S,k)$, and $\rho'(S,k)$ are closely related to our inverted index ADT of Chapter 2. For individual query terms these methods may be defined using that ADT, as shown in Figure 5.13 for $\tau(t,k)$ and $\tau'(t,k)$. Thus, for the inverted index of Figure 2.1 (page 34) we have

$$\tau(\text{“first”}, 745466) = [745501, 745501]. \tag{5.39}$$

```
    τ(A ... B,k) ≡
1     if k = ∞ then
2       return [∞,∞]
3     if k = −∞ then
4       return [−∞,−∞]
5     [u,v] ← τ(A,k)
6     if [u,v] = [∞,∞] then
7       return [∞,∞]
8     [u',v'] ← τ(B,v+1)
9     if [u',v'] = [∞,∞] then
10      return [∞,∞]
11    [u'',v''] ← τ'(A,u'−1)
12    return [u'',v']

    τ(A ◁ B,k) ≡
13    if k = ∞ then
14      return [∞,∞]
15    if k = −∞ then
16      return [−∞,−∞]
17    [u,v] ← τ(A,k)
18    if [u,v] = [∞,∞] then
19      return [∞,∞]
20    [u',v'] ← ρ(B,v)
21    if [u',v'] = [∞,∞] then
22      return [∞,∞]
23    if u' ≤ u then
24      return [u,v]
25    else
26      return τ(A ◁ B,u')

    ρ(A ▷ B,k) ≡
27    if k = ∞ then
28      return [∞,∞]
29    if k = −∞ then
30      return [−∞,−∞]
31    [u,v] ← ρ(A,k)
32    if [u,v] = [∞,∞] then
33      return [∞,∞]
34    [u',v'] ← τ(B,u)
35    if [u',v'] = [∞,∞] then
36      return [∞,∞]
37    if v' ≤ v then
38      return [u,v]
39    else
40      return ρ(A ▷ B,v')
```

Figure 5.14 Implementation of $\tau(A \text{ ... } B, k)$, $\tau(A \lhd B, k)$, and $\rho(A \rhd B, k)$, where A and B are GC-lists.

The definition of $\rho(t, k)$ is similar to that of $\tau(t, k)$; the definition of $\rho'(t, k)$ is similar to that of $\tau'(t, k)$. For fixed-size intervals $[i]$, implementation of the GC-list methods is even more straightforward, because the solution may be computed directly from k (see Exercise 5.5).

The methods for the binary operators are built upon the methods of their operands. Figure 5.14 provides some examples. In the implementation of $\tau(A \text{ ... } B, k)$, most of the lines handle boundary cases associated with ∞ and $-\infty$. The core of the algorithm is expressed by lines 5, 8, and 11, reflecting the idea illustrated by Figure 5.12. On line 5, the first interval in A starting at or after k is computed. On line 8, the first interval in B starting at or after the interval from A is computed. The end position of this interval is the end position of the solution to $\tau(A \text{ ... } B, k)$. Finally, on line 11 the start position of the solution is computed.

The methods are amenable to implementation by galloping search (see Figure 2.5 on page 42). To generate all solutions to a query Q, we call $\tau(Q, k)$ iteratively:

```
k ← 0
while k < ∞ do
    [u,v] ← τ(Q,k)
    if k ≠ ∞ then
        output [u,v]
        k ← u + 1
```

5.3 Further Reading

At the beginning of Section 5.1, we implied that a query processor for ranked retrieval has to follow either the conjunctive or the disjunctive Boolean model. However, combinations are possible. For example, it is possible to initially evaluate the query according to a conjunctive interpretation and then switch to a disjunctive interpretation if the conjunctive one matches too few documents. A more general version of this "weak-AND" approach is described by Broder et al. (2003).

The MaxScore heuristic (Section 5.1.1) is due to Turtle and Flood (1995). A similar algorithm was proposed in earlier work by Smith (1990). More recently, Strohman et al. (2005) presented an improved version of MaxScore that uses precomputed topdocs lists (you may think of these lists as a very aggressively pruned index) to obtain a lower bound for the score of the kth-best search result. This lower bound can then be used to remove query terms from the heap before a single document is scored. Strohman et al. (2005) report time savings of 23% compared to the original MaxScore algorithm. Zhu et al. (2008) propose a variation of MaxScore that can be used for ranking functions that have a term proximity component. Their method is similar to Strohman's approach but uses a phrase index to compute proximity-aware topdocs.

Impact ordering (Section 5.1.4) was first studied by Persin et al. (1996) under the label "frequency-sorted indexes". Refinements of this basic method were explored in a series of publications by Anh et al. (2001, 2004, 2006). The final paper in this series is of particular interest, because it demonstrates the suitability of the impact-ordered index organization for document-at-a-time query processing strategies — something that is not immediately obvious.

The Pat region algebra (Gonnet, 1987; Salminen and Tompa, 1994), initially developed to meet the needs of the *New Oxford English Dictionary* project, was later commercialized by Open Text Corporation as an integral part of their search engine (Open Text Corporation, 2001). Tim Bray, the sole author of the first version of that engine, went on to become a co-creator of the XML Standard. Later versions of the engine provided search services for Yahoo! during the mid-1990s and continue to power the company's enterprise content management products.

The success of Pat inspired a number of extensions and improvements. The PADRE system (Hawking and Thistlewaite, 1994) provided a parallel implementation of Pat. Burkowski (1992) presents containment and set operators for a hierarchical region algebra. The region algebra presented in this chapter is based on Clarke et al. (1995a, 1995b) and may be viewed as an extension (and simplification) of Burkowski's region algebra. Dao et al. (1996) and Jaakkola and Kilpeläinen (1999) present further extensions in support of recursive structure (e.g., speeches that contain speeches). Consens and Milo (1995) explore theoretical aspects and limitations of region algebras. The region algebra described by Navarro and Baeza-Yates (1997) organizes regions into multiple hierarchies, supporting direct ancestor/child relationships and recursive structure.

Additional information on the efficient implementation of the combination operators can be found in Clarke and Cormack (2000). Zhang et al. (2001) discuss algorithms for efficiently implementing containment queries in the context of a relational database system. Young-Lai and Tompa (2003) describe a bottom-up, one-pass approach to implementation. Boldi and Vigna (2006) explore efficient implementation through lazy evaluation.

5.4 Exercises

Exercise 5.1 Equation 5.12 on page 142 states that the worst-case complexity of the document-at-a-time algorithm with heaps is $\Theta(N_q \cdot \log(n) + N_q \cdot \log(k))$.

(a) Characterize the kind of input (i.e., document score distribution) that represents the worst case.

(b) Prove that, on average, the complexity of the algorithm is strictly better than $\Theta(N_q \cdot \log(k))$. You may assume that $k > n$ and that document scores are distributed uniformly, that is, every document is equally likely to have the highest score, second-highest score, and so forth.

Exercise 5.2 Design a term-at-a-time query processing algorithm for conjunctive queries (Boolean AND) that supports accumulator pruning.

Exercise 5.3 Figure 5.7(b) shows that the average CPU time per query drops sharply if less than $B = 3$ bits are used per precomputed score contribution. This is a side effect of the MaxScore heuristic employed by the query processing algorithm. Explain why MaxScore is more effective for smaller values of B.

Exercise 5.4 Given the assumptions of Section 5.2.3, express the following queries in the region algebra:

(a) Find plays that contain "Birnam" followed by "Dunsinane".

(b) Find fragments of text that contain "Birnam" and "Dunsinane".

(c) Find plays in which the word "Birnam" is spoken by a witch.

(d) Find speeches that contain "toil" or "trouble" in the first line, and do not contain "burn" or "bubble" in the second line.

(e) Find a speech by an apparition that contains "fife" and that appears in a scene along with the line "Something wicked this way comes".

Exercise 5.5 Write pseudo-code implementing the four GC-list methods for fixed-length intervals: $\tau([i], k)$, $\rho([i], k)$, $\tau'([i], k)$, and $\rho'([i], k)$.

Exercise 5.6 Write pseudo-code implementing the GC-list methods for the two projection operators: $\tau(\pi_1(A), k)$, $\rho(\pi_1(A), k)$, $\tau'(\pi_1(A), k)$, $\rho'(\pi_1(A), k)$, $\tau(\pi_2(A), k)$, $\rho(\pi_2(A), k)$, $\tau'(\pi_2(A), k)$, and $\rho'(\pi_2(A), k)$.

Exercise 5.7 Following the pattern of the algorithms shown in Figure 5.14 (page 168), write pseudo-code for the following methods:

(a) $\rho(A \text{ ... } B, k)$
(b) $\tau(A \bigtriangleup B, k)$
(c) $\tau(A \bigtriangledown B, k)$
(d) $\tau(A \not\triangleright B, k)$
(e) $\rho(A \lhd B, k)$
(f) $\tau'(A \lhd B, k)$

Exercise 5.8 Write pseudo-code implementing the four GC-list methods for phrases: $\tau(t_1...t_n, k)$, $\rho(t_1...t_n, k)$, $\tau'(t_1...t_n, k)$, and $\rho'(t_1...t_n, k)$. (*Hint:* As a starting point, consider the implementation of the **nextPhrase** function in Figure 2.2 on page 36).

Exercise 5.9 (project exercise) Implement the BM25 ranking formula (Equation 8.48 on page 272), using a document-at-a-time evaluation strategy. Your implementation should use the MaxScore heuristic to reduce the number of document scores computed.

5.5 Bibliography

Anh, V. N., de Kretser, O., and Moffat, A. (2001). Vector-space ranking with effective early termination. In *Proceedings of the 24th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 35–42. New Orleans, Louisiana.

Anh, V. N., and Moffat, A. (2004). Collection-independent document-centric impacts. In *Proceedings of the 9th Australasian Document Computing Symposium*, pages 25–32. Melbourne, Australia.

Anh, V. N., and Moffat, A. (2006). Pruned query evaluation using pre-computed impacts. In *Proceedings of the 29th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 372–379. Seattle, Washington.

Boldi, P., and Vigna, S. (2006). Efficient lazy algorithms for minimal-interval semantics. In *String Processing and Information Retrieval, 13th International Conference*, pages 134–149. Glasgow, Scotland.

Broder, A. Z., Carmel, D., Herscovici, M., Soffer, A., and Zien, J. (2003). Efficient query evaluation using a two-level retrieval process. In *Proceedings of the 12th International Conference on Information and Knowledge Management*, pages 426–434. New Orleans, Louisiana.

Burkowski, F. J. (1992). An algebra for hierarchically organized text-dominated databases. *Information Processing & Management*, 28(3):333–348.

Büttcher, S., and Clarke, C. L. A. (2006). A document-centric approach to static index pruning in text retrieval systems. In *Proceedings of the 15th ACM International Conference on Information and Knowledge Management*, pages 182–189. Arlington, Virginia.

Carmel, D., Cohen, D., Fagin, R., Farchi, E., Herscovici, M., Maarek, Y., and Soffer, A. (2001). Static index pruning for information retrieval systems. In *Proceedings of the 24th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 43–50. New Orleans, Louisiana.

Carpineto, C., de Mori, R., Romano, G., and Bigi, B. (2001). An information-theoretic approach to automatic query expansion. *ACM Transactions on Information Systems*, 19(1):1–27.

Clarke, C. L. A., and Cormack, G. V. (2000). Shortest-substring retrieval and ranking. *ACM Transactions on Information Systems*, 18(1):44–78.

Clarke, C. L. A., Cormack, G. V., and Burkowski, F. J. (1995a). An algebra for structured text search and a framework for its implementation. *Computer Journal*, 38(1):43–56.

Clarke, C. L. A., Cormack, G. V., and Burkowski, F. J. (1995b). Schema-independent retrieval from heterogeneous structured text. In *Proceedings of the 4th Annual Symposium on Document Analysis and Information Retrieval*, pages 279–289. Las Vegas, Nevada.

Consens, M. P., and Milo, T. (1995). Algebras for querying text regions. In *Proceedings of the 14th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 11–22. San Jose, California.

Dao, T., Sacks-Davis, R., and Thom, J. A. (1996). Indexing structured text for queries on containment relationships. In *Proceedings of the 7th Australasian Database Conference*, pages 82–91. Melbourne, Australia.

Gonnet, G. H. (1987). PAT *3.1 — An Efficient Text Searching System — User's Manual*. University of Waterloo, Canada.

Hawking, D., and Thistlewaite, P. (1994). Searching for meaning with the help of a PADRE. In *Proceedings of the 3rd Text REtrieval Conference (TREC-3)*, pages 257–267. Gaithersburg, Maryland.

Jaakkola, J., and Kilpeläinen, P. (1999). *Nested Text-Region Algebra*. Technical Report CC-1999-2. Department of Computer Science, University of Helsinki, Finland.

Lester, N., Moffat, A., Webber, W., and Zobel, J. (2005). Space-limited ranked query evaluation using adaptive pruning. In *Proceedings of the 6th International Conference on Web Information Systems Engineering*, pages 470–477. New York.

Moffat, A., and Zobel, J. (1996). Self-indexing inverted files for fast text retrieval. *ACM Transactions on Information Systems*, 14(4):349–379.

Navarro, G., and Baeza-Yates, R. (1997). Proximal nodes: A model to query document databases by content and structure. *ACM Transactions on Information Systems*, 15(4):400–435.

Ntoulas, A., and Cho, J. (2007). Pruning policies for two-tiered inverted index with correctness guarantee. In *Proceedings of the 30th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 191–198. Amsterdam, The Netherlands.

Open Text Corporation (2001). *Ten Years of Innovation.* Waterloo, Canada: Open Text Coporation.

Persin, M., Zobel, J., and Sacks-Davis, R. (1996). Filtered document retrieval with frequency-sorted indexes. *Journal of the American Society for Information Science*, 47(10):749–764.

Salminen, A., and Tompa, F. W. (1994). PAT expressions — An algebra for text search. *Acta Linguistica Hungarica*, 41(1–4):277–306.

Smith, M. E. (1990). *Aspects of the P-Norm Model of Information Retrieval: Syntactic Query Generation, Efficiency, and Theoretical Properties.* Ph.D. thesis, Cornell University, Ithaca, New York.

Strohman, T., Turtle, H., and Croft, W. B. (2005). Optimization strategies for complex queries. In *Proceedings of the 28th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 219–225. Salvador, Brazil.

Turtle, H., and Flood, J. (1995). Query evaluation: Strategies and optimization. *Information Processing & Management*, 31(1):831–850.

Young-Lai, M., and Tompa, F. W. (2003). One-pass evaluation of region algebra expressions. *Information Systems*, 28(3):159–168.

Zhang, C., Naughton, J., DeWitt, D., Luo, Q., and Lohman, G. (2001). On supporting containment queries in relational database management systems. In *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data*, pages 425–436. Santa Barbara, California.

Zhu, M., Shi, S., Yu, N., and Wen, J. R. (2008). Can phrase indexing help to process non-phrase queries? In *Proceedings of the 17th ACM Conference on Information and Knowledge Management*, pages 679–688. Napa, California.