Static Score Bucketing in Inverted Indexes

Chavdar Botev
Cornell University
4130 Upson Hall
Ithaca, NY, USA
cbotev@cs.cornell.edu

Nadav Eiron, Marcus Fontoura, Ning Li, Eugene Shekita
IBM Almaden Research Center
650 Harry Rd
San Jose, CA, USA
trevi@almaden.ibm.com

ABSTRACT

Maintaining strict static score order of inverted lists is a heuristic used by search engines to improve the quality of query results when the entire inverted lists cannot be processed. This heuristic, however, increases the cost of index generation and requires complex index build algorithms. In this paper, we study a new index organization based on static score bucketing. We show that this new technique significantly improves in index build performance while having minimal impact on the quality of search results.

Categories and Subject Descriptors H.3[Information Systems]: Information Storage and Retrieval; E.1[Data]: Data Structures

General Terms: Algorithms, Performance

Keywords: Indexing, Search engines, Static scoring

1. INDEXING WITH STATIC SCORE BUCKETING

In previous work, Long and Suel [6] have proposed an inverted lists organization that is based on a static rank order of the postings. Such an organization improves the result quality when the entire index cannot be scanned. However, it degrades index build performance and increases the complexity of index build algorithms [3]. This issue stems from the fact that a change in the score of a single document may translate to many updates to postings lists.

In this work, we propose relaxing the total static score order to a partial order. The partial order groups postings with similar static scores into buckets. Thus, we only need to maintain order across buckets and we allow the document order inside of a bucket to be arbitrary. For example, postings inside a bucket can be stored in an increasing docid (document identifier) order, which need not be the same as the static score order. This allows for an efficient query evaluation, since posting lists can be joined, and for efficient indexing algorithms, since a full sort on the static scores can be avoided. Although this approach can lead to slight degradation in the quality of the returned results, we experimentally show that the degradation in quality is controllable and can be made negligible.

Copyright is held by the author/owner.
CIKM'05, October 31–November 5, 2005, Bremen, Germany.
ACM 1-59593-140-6/05/0010.

2. INDEXING ALGORITHM

Our indexing algorithm is based on a re-merge index-update strategy, which Lester et al. [5] found to have good performance. In the re-merge strategy, new documents are added to a delta index, which can be an in-memory index. When the delta index gets to a certain size it is merged with the main index to produce the a new main index and the delta index is reset. The challenge is that using static rank to dictate inverted list order presents problems when merging indexes, especially when the docid order is the same as the static score order [3]. Since docids need to be changed to reflect new static scores, a full sort of the output posting lists is required to reflect the new scores in new main index. For instance, when a single document with a high score is added to the system, most of the docids become invalid.

We now present the re-merge algorithm based on static score bucketing. The algorithm has a time complexity that is linear in the combined size of the main and delta indexes. Our algorithm uses the following in-memory structures: (i) *remList*: the list of documents that need to be removed, and (ii) *changeTable*: a mapping from the docid of documents that changed their bucket to their new bucket number. These structures can be readily stored in memory.

```
   Function IndexMerge
1   for each Term t {
2     bucketPages = new Page[numberOfbuckets];
3     main[] = mainIndex.getPostings(t);
4     delta[] = deltaIndex.getPostings(t);
5     while (exists main[] and exists delta[]) {
6       curDocid = min docid from main[] and delta[];
7       if (remList.contains(curDocid)) continue;
8       if (changeTable.contains(curDocid))
9         curBucket = changeTable.lookup(curDocid);
10        else curBucket = current bucket for curDocid;
11        addToBucket(curBucket, curDocid, bucketPages);
12    } //end while (main merge loop);
13  } //end for
```

The algorithm merges, for each term t, the corresponding postings lists from the delta and the main indexes: *delta[]* and *main[]* arrays respectively. The size of these arrays is the number of buckets and *delta[]* and *main[]* contain one posting list per bucket for the given term t. These posting lists are sorted by docid. Each iteration of the main loop processes the posting with the minimum docid (line 6). If its

document has been deleted, that posting entry is not added to the merged index (line 7). The output postings are partitioned into several output streams (the *bucketPages* array) based on the static score bucket of the posting document. These output stream are represented by buffer pages, which are flushed to disk whenever the they fill up. If the document in question is in the *changeTable*, it is redirected to the output stream corresponding to the new static score bucket (line 8). Otherwise, it remains on the output stream that corresponds to its old bucket (line 11).

3. EXPERIMENTAL RESULTS

We performed two main sets of experiments. The first set tested the index build performance. The second set of experiments tested the change in quality of the results when using inverted lists bucketing. For both sets of experiments, results were compared to the baseline system with documents completely ordered by their static scores.

Experimental setup. Our scheme introduces several tunable parameters that potentially influence performance. The *bucket type* parameter specifies the function used for bucketing. We follow Haveliwala [4] who describes static score quantization methods in the related problem of finding efficient encodings of static scores. For our experiments, we used the following non-linear quantizations: logarithmic $G(x) \propto \log x$, square root $G(x) \propto \sqrt(x)$, exponential $G(x) \propto x^b$ and equi-depth where each bucket contains approximately the same number of documents. We also experimented with an additional bucketing scheme, which we refer to as adaptive. It groups documents so that the maximum difference of the static scores within a bucket does not exceed a fixed value. A second parameter that we tested is the number of buckets per inverted list. Finally, we experimented with two types of static score methods: in-degree (the number of hosts referring to the document) [3] and PageRank (a measure of the popularity based on the random walk model [1]).

We performed our experiments with a real-world query load from the IBM intranet: a sample of 277 unique queries out of 116,313 total queries. All the experiments were executed using the Trevi intranet search engine [3].

Experiments on indexing performance. For these experiments, we implemented the index re-merge algorithm described in Section 2. We compared the proposed algorithm with a re-merge algorithm based on strict static score ordering. We built an index with four buckets using all the bucket types previously described. The results for all the bucketing methods we tested were similar.

We fixed a main index for 500K documents and varied the delta index size, from 64,000 documents to 128,000 documents. The results indicated that using static score bucketing we can achieve more than 20% increase in index build performance to an already highly-optimized indexing algorithm. Furthermore, the results are independent of the size of the delta index being merged. Thus, a search engine using static score bucketing can have an increased indexing throughput. Therefore, the search engine can update its index more frequently, resulting in more up-to-date content provided to the users.

Experiments on query-results quality. These experiments illustrate how the query results change when we apply static score bucketing to the inverted lists. We tested how the number of inversion between the result rankings (the Kendal Tau similarity measure [2]) varies with the number of scanned documents. The ground truth for our experiments were the results returned by the search engine using its standard total ordering of t he inverted lists based on the static score, and scanning the entire inverted lists (i.e. no "early-termination").

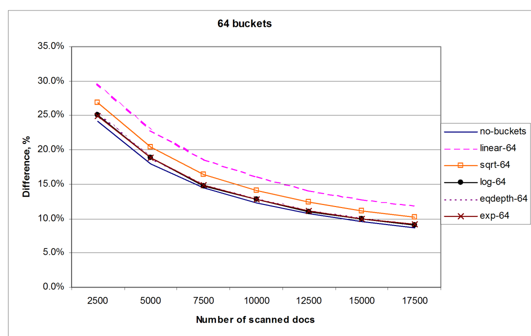

Figure 1: Static score = in-degree, Number of buckets = 64

Figure 1 presents the experiments using in-degree static score and 64 buckets. The experiment compares the different bucketing types when the number of scanned postings increases. As a reference, we have also included the results with early termination when no bucketing is applied (the "no-buckets" series).

As it can be expected, the difference to the ground truth quickly decreases when the engine scans larger potions. The comparison to the reference "no-buckets" series shows that most of this difference can be attributed to the early termination and not to the bucketing type. Furthermore, the figure shows that the differences between the various bucketing types are small. Experiments with larger values for the number of documents scanned (not presented due to lack of space) show that the differences get even smaller when larger potions of the posting lists are scanned.

The experiments with PageRank show similar trends and we do not present them due to lack of space. The differences between the tested bucketing schemes were even smaller due to the higher precision of the PageRank scores, which allows better clustering of the postings into buckets.

4. REFERENCES

[1] S. Brin and L. Page. The anatomy of a large-scale hypertextual (web) search engine. *Computer Networks and ISDN Systems*, 30(1–7):107–117, 1998.

[2] R. Fagin, R. Kumar, and D. Sivakumar. Comparing top k lists. *SIAM Journal on Discrete Mathematics*, 17(1):134–160, 2003.

[3] M. Fontoura, A. Neumann, S. Rajagopalan, E. Shekita, and J. Zien. High performance index build algorithms for intranet search engines. In *VLDB' 2004*.

[4] T. Haveliwala. Efficient encoding for document ranking vectors. In *Proc. of 4th Int. Conference on Internet Computing*, 2003.

[5] N. Lester, J. Zobel, and H. E. Williams. In-place versus re-build versus re-merge: index maintenance strategies for text retrieval systems. In *CRPIT '2004*.

[6] X.Long and T. Suel. Optimized query execution in large search engines with global page ordering. In *Proc. of the 29th Int. Conf. on Very Large Databases*, 2003.