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

November 23, 2005

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 time-consuming 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. We also provide upper bounds on the quality degradation and verify experimentally the benefits of the proposed approach.

1 Introduction

With the web becoming the preferred medium for storage and dissemination of information, search engines are faced with the problem of indexing and searching large collections of documents. On the other hand, user interactivity requirements keep increasing as well. These two trends combine to force a search engine to be able to return results without completing a full scan of index data structures for some queries. It is therefore crucial that index data structures will allow for efficient partial scans that do not miss any highly important documents.

Long and Suel [22] have proposed an index organization that is based on a static rank order of the documents. The main benefit of such an organization is

the improvement in result quality when the entire index cannot be scanned. Intuitively, the static rank, such as Google's PageRank [3], reflects the document's quality or importance. In this index organization, higher-ranking documents are stored in the beginning of the index and are more likely to be accessed and considered as results. On the other hand, this index organization degrades index build performance and increases the complexity of index build algorithms [10].

In this paper, we provide a study of an alternative organization of the inverted postings lists. It relaxes some of the stringent requirements of a total static rank order of the index by imposing only a partial order. Documents are grouped in buckets and the order of the documents within a bucket is arbitrary. Although this approach can lead to slight degradation in the quality of the returned results, we show that there are significant improvements of the index build time. We also experimentally show that the degradation in quality is controllable and can be made negligible. The main contributions of this paper are:

- A new index organization based on static rank bucketing.
- An indexing algorithm that leverages the above index organization and leads to significant improvements in index build time.
- A formal analysis of the maximum expected degradation in result set quality as a result of static score bucketing.
- An experimental study of the influence of bucketing on indexing performance, showing that the new indexing algorithm can indeed speed up the index generation.
- An experimental study on the influence of bucketing on the the quality of the produced query results. We demonstrate that we can achieve significant benefits in index build time at the cost of only a minor decrease in result quality.

2 Preliminaries

The key task of information retrieval is finding documents relevant to a user query. Usually, relevance is measured in terms of one or more numeric *scores*. Higher scores denote more relevant documents. There has been an extensive research on methods for calculating scores [2, 7]. In general, two kind of scores may be distinguished: *dynamic scores* and *static scores*. A dynamic score represents how well a document matches a user query. An example of such score is the popular cosine similarity of $tf \times idf$ vectors [18].

Static scores, on the other hand, represent the general popularity or value of a document. Static scores are especially common in linked environments like the Web or scientific publication, where a link from one document to another is considered a transfer of authority. Some of the static scoring methods in linked environments are PageRank [3] and in-degree, which is simply the number of

different documents (or hosts, in the Web case) that contain a link to a given document [7]. Static scores do not depend on the query and can thus be precomputed. Search engines use a combination of static and dynamic scores to produce a list of documents in decreasing order of their scores. The list is usually referred to as a *ranked list*. The position of the document in the list is its *rank*. Lower ranked documents will have higher scores.

An *inverted index* [13] is a popular indexing structure that allows high-performance full-text search. An inverted index has a set of *inverted lists*, one per term in the document corpus. Each inverted list (or postings list), in turn, is a sequence of *postings* that describe the occurrences of the term in the document collection. Usually, each posting describes the term's occurrences within a single document by list of all positions within the document containing the term.

An important aspect of inverted lists organization is the order of postings in the list. Traditionally, the postings are in an increasing order of their document identifiers *(docids)*. This organization allows an efficient evaluation of conjunctive keyword queries because it allows single-pass processing similar to a merge join on the docid. Furthermore, if the docids monotonically increase for every new document, insertions in the document collection can be handled by appending postings at the end of the inverted lists.

Another possible organization is ordering based on a static score of the documents. This facilitates efficient evaluation techniques for top-k queries. For example, Fagin's Threshold Algorithm [9] can utilize the decreasing score order to allow evaluation of top-k queries without scanning the entire inverted lists. Static-score ordering is particularly important in the case when the entire posting lists cannot be processed. This is typical for search engines where the amount of data can lead to very large postings lists and high query loads. As shown in [22], maintaining postings order based on a decreasing static score can improve the quality of the returned results. When this technique is employed we may treat the docid's as reflecting the rank, and leave the index ordered by docid [10].This requires only minimal changes to the index build algorithm.

In this paper we study the grouping of inverted list postings with similar static score into *static score buckets* and its effects on query performance and results. The order of the postings within a bucket is not important. Thus, this is a relaxation of the original requirement that all postings are in a decreasing order of their static score.

3 Index Build

In this section, we present an algorithm that uses static score bucketing for efficiently generating the inverted index. While effective for IR systems, inverted indexes are not efficient for handling updates. This is primarily because a change in a single document may translate to many updates to postings lists – one for each term appearing in the document. Previous work on build algorithms for inverted indexes can be classified into the following three categories [15]:

- In-place: when a new document is added to the index, the inverted lists

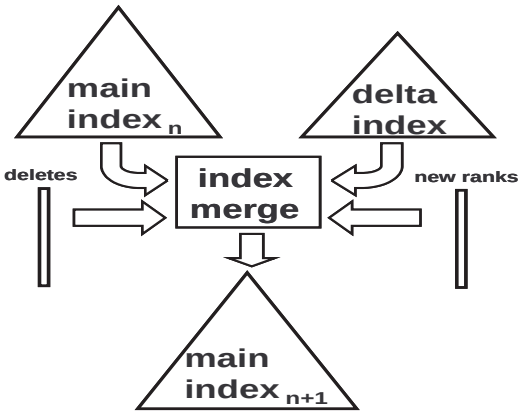


Figure 1: Index re-merge algorithm

for each of its terms are updated in-place. Obviously, special care should be taken to minimize the number of disk accesses.

- Re-merge: new documents are added to a delta index, which could 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. At that time the delta index is reset.

- Re-build: this strategy discards any existing index and scans the entire document collection to create the new index.

Lester et al. [15] found that, in general, the re-merge strategy has the best performance. In the remainder of this section we focus on the re-merge strategy and show how static score bucketing can improve its performance. Figure 1 illustrates the basic index re-merge algorithm. It takes as inputs the main index at generation n, the current delta index, the list of deleted documents, and a table with the new ranks. For each term in the corpus, the posting lists for the main and delta indexes are merged to produce a single posting list in the main index at generation $n + 1$.

Unfortunately, using static rank to dictate inverted list order presents additional problems when merging indexes. Since docid's need to be changed to reflect new static ranks, a full sort of the output posting lists is required to reflect the new ranks in main index $n+1$. For instance, when a single document with a high score is added to the system, most of the docids become invalid.

To alleviate this problem, we propose relaxing the total static score order to a partial order. The partial order groups postings with similar static scores into buckets. By splitting the postings lists into buckets, we only need to maintain order across buckets, but we allow the document order inside of a bucket to be arbitrary. This does not mean it can be random – postings are still stored inside

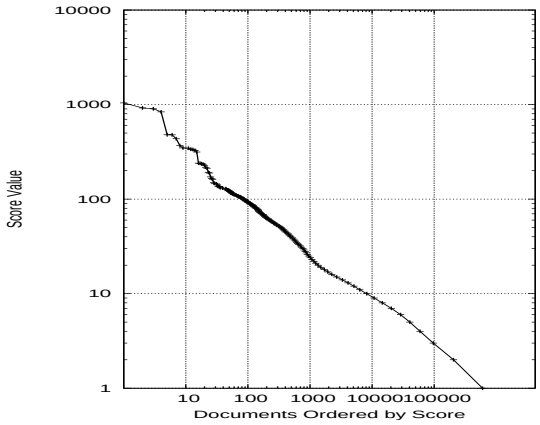


Figure 2: Static score distribution inside the IBM intranet

a bucket in increasing docid order. However, this ordering, within a bucket, may be in disagreement with the static score ordering. This allows for efficient query evaluation, since posting lists can be joined, and for efficient index re-merge algorithm, since the sort on the docids can be avoided as described in the next section.

It has been shown [10, 23] that link-based static scores tend to be stable and that they typically follow a power-law distribution [10]. Figure 2 is a log-log graph that shows the distribution of the static scores in the IBM intranet, where only 10% of the documents have very high static scores. The static score measure used in Figure 2 is host in-degree, which is the number of links from different hosts that refer to the document [10].

The above scheme that orders documents by score buckets, instead of strict ranks, should therefore need a very small number of buckets (due to the power-law distribution of the scores) to provide reasonable quality. Moreover, each of these buckets should be very stable, i.e., documents should rarely move from one bucket to another, making index merge cheaper.

3.1 Index Build Algorithm

We now describe the re-merge algorithm based on static score bucketing. It merges an existing main index with a delta index that contains the changes to the document collection. Our algorithm uses the following global variables:

mainIndex the main index being merged.

deltaIndex the delta index being merged. In practice, there could be a series of delta indexes. For simplicity we consider a single delta index. Extending the algorithm to handle the general case is straightforward.

```
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))
 8        continue;
10      if (changeTable.contains(curDocid))
11        //move the postings for the current docid
12        //to the temporary space for the new bucket
13        curBucket = changeTable.lookup(curDocid);
14      else
15        curBucket = current bucket for curDocid;
16      addToBucket(curBucket, curDocid, bucketPages);
17    } //end while (main merge loop);
18  } //end for
```

Figure 3: Index Merge Algorithm

remList contains the list of documents that need to be removed.

changeTable a table containing information for documents that changed their bucket. It maps a docid to the number of the new bucket.

In both the main and in the delta indexes the posting list for each term t is sorted by $<bucketNumber, docid>$, where small bucket numbers contain the documents with higher static scores and the docids are simply assigned by the order the documents are added to the indexes. Using this scheme, whenever a document is updated it gets a new docid (with the current document count value) and the old docid is deleted.

The algorithm described in Figure 3 merges, for each term t, the corresponding postings lists from the delta and the main indexes. We open one cursor per bucket in both the main and delta indexes. This is easy to do since the bucket number is part of the index key and therefore access to each bucket is efficient. The main index cursors are stored in the *main[]* array while the cursors for the delta index are stored in the *delta[]* array. Each iteration of the main loop processes the posting with the minimum docid among all the buckets in both the main and delta indexes (line 6). If that document has been deleted that posting entry is not added to the merged index (lines 7-8).

The output postings are partitioned into several output streams (the *bucketPages* array) based on the static score bucket of the posting. These output streams are represented by buffer pages. Since postings are added to the output pages in docid order the output index will also be sorted by $<bucketNumber, docid>$. If the document in question is in the *changeTable*, it is redirected to the output stream corresponding to the new static score bucket (lines 10-13). Otherwise, it remains on the output stream that corresponds to its old bucket (lines 14-13). The posting is physically added to its corresponding buffer in method *addTo-*

Bucket (line 16), which takes care of flushing the buffers to disk whenever the they fill up.

It is easy to show that the above index re-merging algorithm has a time complexity that is linear in the combined size of the main and delta indexes. Algorithms that use the static rank as the docid, on the other hand, require a full sort on the docid part of the key for each posting list in the main and delta indexes. The *remList* and the *changeTable* can be kept readily in memory. For example, the IBM intranet changes at the rate of about 500,000 documents daily [10]. Thus, these structures will occupy at most several megabytes in the worst case when all the modifications involve change of static score bucket. Even in a larger scale application, such as Internet search, the size of those data structures is significantly smaller than the size of other structures required for index build (such as those required to calculate the static scores, etc.). Therefore, the presented algorithm is very efficient both in terms of time and memory complexity, as long as the static scores are stable, relative to the chosen bucket size.

One important observation is that if the index relies on delta compression of the docid's, this index organization might generate larger indexes since the index key for each posting list is now the pair $<bucketNumber, docid>$, instead of only docid. In our implementation compression is not used for at the docid level, but only for the positions of a term within a document. Therefore, the index build algorithm proposed here did not have any effect in index size for our implementation and we do not show experimental results on index sizes.

3.2 Bucketing Schemes

One of the choices that obviously needs to be made with a bucketized indexing scheme is the mapping of static scores into buckets. One can view the process of bucketizing static score as a quantization of the static score into coarser granularity. In this section, we provide background information on methods for the quantization of scalar values (see [11] for an excellent overview). These methods will later be used for testing various methods of static-score bucketing.

The simplest method is *linear uniform quantization* where the size of all buckets is the same. Usually, the number of buckets is of the form 2^N. Thus, the bucket number can be determined by the N most-significant bits (MSB) of the score. The problem with uniform quantization is it does not account for any distortions of the data. To address this, the theory of non-uniform quantization has been developed. It has been proved that non-uniform quantization can be represented using companders. *Companders* are based on smooth monotonic non-linear "compressor" function $G(x)$. The process has two phases. The compression phase finds the bucket for a value x: $b = u(G(x))$, where $u(\cdot)$ is the uniform quantization function. The expansion phase computes an estimator of the original value: $\hat{x} = G^{-1}(b) = G^{-1}(u(G(x)))$.

There are various options for the compressor function. We follow Haveliwala [14] who describes static score quantization methods in the related problem of finding efficient encodings of static scores. The quantization schemes he uses are the linear (uniform) quantization and the following non-linear quantiza-

tions: 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.

4 Querying and Ranking

Next, we focus on the effects of static score bucketing on result ranking. Specifically, we prove the relationship between the expected degradation in the quality of query results and the bucketing parameters used. As we discussed in Section 2, the final result ranking is based on a total score that is associated with each potential result. In our case, we assume that the final score is an aggregation of dynamic score, which is query dependent, and a static score, which is query-independent. Query processing itself remains unchanged, with the documents being ordered by the key (bucket number, docid) instead of just by docid.

In the rest of the exposition, we will use the following formal notation. Let $\mathcal{D}$ be the enumerable set of indexed documents. Let $S : \mathcal{D} \rightarrow [0, M_S]$ be a function that returns the static score of a document, and let $B : \mathcal{D} \rightarrow \mathcal{N}$ be the function that maps a document to its static score bucket. For linear bucketing, $B(d) = u_N(S(d))$, where $u_N(\cdot)$ is the linear transformation taking the N most significant bits. For non-linear bucketing with compressor function G, $B(d) = u_N(G(S(d)))$. A ranking $r : \mathcal{D} \rightarrow \mathcal{N} \cup \infty$ is a function that maps a document d to its position in the result list or ∞, if the document is not part of the top-k results. Finally, an inverted list $I : \mathcal{N} \rightarrow \mathcal{D}$ is a mapping from a position to a document.

Let us consider the partial order on documents that is induced by bucketing, namely $\preceq_B \subseteq \mathcal{D} \times \mathcal{D}$ such that $d_1 \preceq_B d_2$ iff $B(d_1) \leq B(d_2)$. The following lemma holds.

Lemma 1 *Any inverted list i using static score order is a refinement of $\prec_B$.*

Obviously, any inverted list i' that uses any bucketized static score order is also a refinement of $\preceq_B$. Thus, we have:

Lemma 2 *Let a i be an inverted list with static score order and i' be an inverted list with static score bucketing. If the bucket with number n comprises the postings with positions from l_n to u_n, then $i'(l_n), i'(l_n + 1), ..., i'(u_n)$ is a permutation of $i(l_n), i(l_n + 1), ..., i(u_n)$.*

Proof Outline. It follows from the previous lemma that i and i' are refinements of the same partial order. Therefore, they must agree on the order of documents d_1, d_2 if $d_1 \prec_B d_2$ or $d_2 \prec_B d_1$. Thus, they can only disagree if $B(d_1) = B(d_2)$, i.e. they can order differently only the elements within one bucket. □

Static score bucketing does not change the scores of documents. It only changes their position in the inverted lists. This is in contrast to the effects of score bucketing with the goal of compressing the score information [14, 17]. In

the latter case, the score changes because of the loss of precision. Naturally, the change in score usually causes a change in the final ranked results.

When bucketizing scores, the only cause of perturbations in the final result ranking is because of the effects of early termination. As already discussed, early termination occurs when query engine stops the query evaluation before scanning the entire inverted lists.

Given a query Q, let us consider the result ranking R based on the static score order index I and the result ranking R' based on the static score bucketing index I'. We assume that both inverted indexes are scanned at the same speed, i.e. the early terminations occur at the same position. We are interested in the cases where for some documents d_1 and d_2, $(R(d_1) - R(d_2))(R'(d_1) - R'(d_2)) < 0$. These are the cases where R and R' disagree on the relative order of the documents. However, since the static *score* of documents is identical in I and I' this cannot happen if both documents are scanned.

Theorem 1 *If for some documents $d_1, d_2 \in \mathcal{D}$ both of them are scanned, then $(R(d_1) - R(d_2))(R'(d_1) - R'(d_2)) \geq 0$. In other words, R and R' agree on the order of d_1 and d_2.*

It is also clear that R and R' agree on any pair of documents where neither of them is scanned (such documents will not appear in the result set of either indices). We are therefore only interested in the case where at least one of the indices scans exactly one of the two documents d_1 and d_2. We further consider two variations of this case. In the first variation, each of the indices scanned just one of the documents (say, w.l.o.g., that n documents were scanned, and $I(d_1) \leq n < I(d_2) \wedge I'(d_2) \leq n < I'(d_1)$). In the second variant one of the indices scanned both the documents, but the other scanned only one.

Let us analyze the probability of the first variant of rank inversion to occur. Observe that if a document has not been scanned by one of the indices but has been scanned by the other, then the document posting is located in the last scanned bucket. Therefore, $B(d_1) = B(d_2)$ and both are equal to the number of the last scanned bucket. If we assume that there is no correlation between the order in which the documents are added to the database and their static score, then the postings from i' in the bucket $B(d_1) = B(d_2)$ are a random permutation of the corresponding postings from i (also, recall Lemma 2). Let $b = r - l + 1$ be the number of elements in the bucket and let $m = n - l + 1$ be the number of postings from the bucket that have been scanned. The probability that this variant occurs is the same as probability that one document posting is among the first m postings, and the other posting is among the remaining $b - m$ postings: $\Pr[b1] = \frac{m}{b} \cdot \frac{b-m}{b}$. Thus, on average there will be $b\frac{m(b-m)}{b^2} = \frac{m(b-m)}{b}$ inversions of of this type. Assuming the evaluation is equally likely to terminate at any position m, the expected average number of inversions is

$$E[b1] \quad = \quad \sum_{m=l}^{b} \frac{1}{b} \frac{m(b-m)}{b} \tag{1}$$

$$= \frac{1}{b^2}\left(\frac{b^2(b+1)}{2} - \frac{b(b+1)(2b+1)}{6}\right) \quad (2)$$

$$= \frac{b^2-1}{6b} \quad (3)$$

As it can be seen, the expected number of inversion grows linearly with the size of the buckets. This confirms the intuitive notion that smaller buckets give more accuracy. In practice, the number of inversions will be lower because the above assumption of independence does not entirely hold. The crawling of documents usually starts from a seed of popular documents with high static score and expands toward less popular documents. Therefore, more popular documents are more likely to be crawled earlier [7]. As a result, the document order within a bucket will be slightly biased toward the static score order.

The analysis of the other case is similar, and is omitted from this extended abstract. It also shows that for a fixed total scoring function, the expected number of inversion grows linearly with the size of the bucket.

Note that not all inversions will result in the difference of the final ranking. Under the top-k evaluation model, only documents that are included in the final top-k results will count toward the difference. It is difficult to compute what this number is because it is query dependent. Sometimes the dynamic score (and consequently, the total score) will be highly correlated with the static score. In these cases, the results will be mostly based on documents from the beginnings of the inverted lists which are most likely to be scanned. Thus, the number of inversions will be low. On the other hand, inversely correlated total and static scores will result in many inversions and the ranking R' will differ significantly. In an attempt to better characterize the actual behavior in practice, we present a detailed experimental study on the influence of static score bucketing on query results.

5 Experimental results

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

5.1 Experimental Setup

Our scheme introduces several tunable parameters that potentially influence performance. These include a bucketing function, the number of buckets generated, and an early termination threshold. The *BucketType* parameter specifies the function used for bucketing. As discussed in Section 4, this function determines which documents are considered to have indistinguishable static score. The choices for this parameter are the ones described in Section 3.2. Certainly,

other (static-score-dependent) bucketing methods are possible. For example, Kraaij et al. [21] propose a bucketing of URL type (e.g. root, sub-root, path, file). We chose not test such bucketing methods because they are not applicable for other static scoring methods.

The *BucketNum* parameter controls the number of buckets per inverted list. Intuitively, the more buckets there are, the closer partial order induced by the bucketing is to the total order. We used only powers of two as values for this parameter.

The *StaticScore* parameter specifies the type of static score method for the documents from the collection. We used two popular static scoring methods: in-degree (the number of hosts referring to the document) [10] and PageRank (a measure of the popularity based on the random walk model [3]). Other static scoring methods are possible but they are not discussed in this paper and are part of our future work.

Another important decision for the quality experiments was the choice of queries to be evaluated. We performed our experiments with a real-world query load from the IBM intranet. The query log contained 116,313 unique queries (from a total of 346,401 queries). As expected, the query frequencies had a typical power-law distribution with a small number of very frequent queries and the majority of the queries having frequency one or two. Using the query log, we created a sample of 277 unique queries. We wanted this sample to represent both frequent and less frequent queries to test our model under varied workload. In creating the query sample, we followed the approach used in [7]. We chose 65 frequent queries (with frequency in the 70th percentile), 112 average queries, and 150 rare queries (frequency 1 or 2). The average length of the inverted list of a query term was about 80000 documents. All the experiments were executed in the context of the Trevi intranet search engine [10]. Trevi's querying runtime is based on the two-level retrieval method of [4].

5.2 Experiments on indexing performance

For the indexing performance experiments, we implemented the index re-merge algorithm described in Section 3. We compared the proposed algorithm with a re-merge algorithm based on strict static rank ordering. We present the result experiments from the linear bucketing scheme with four buckets. The results for the other bucketing methods are similar.

We fixed a main index with roughly one million documents and varied the delta index size, from 125,000 documents to 500,000 documents. The results are presented in Figure 4. The graph shows that using static-score bucketing we can achieve more than a two fold 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 query engine using static-score bucketing can have an increased indexing throughput. Therefore, the engine can update its index more frequently, resulting in more up-to-date content provided to the users.

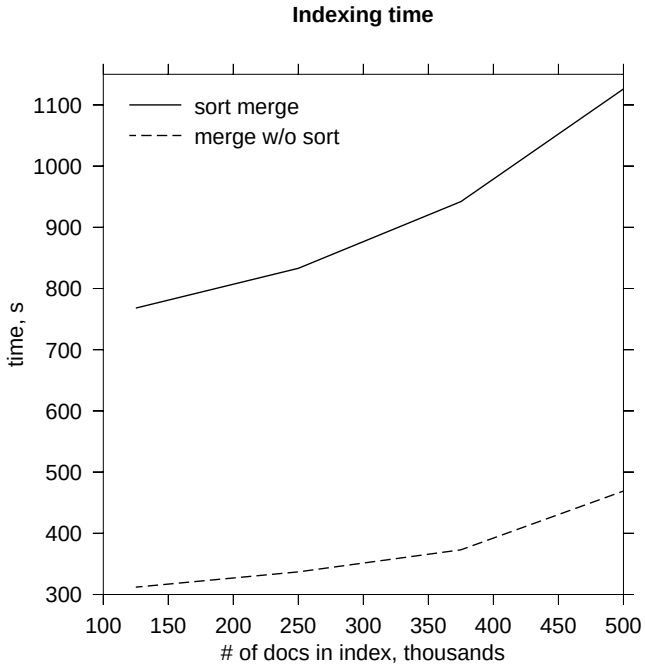


Figure 4: Index re-merge performance

5.3 Experiments on query-results quality

These experiments illustrate how the query results change when we applied static score bucketing to the inverted lists. We show the difference with the ground truth, which for our experiments were the results returned by Trevi using its standard total ordering of the inverted lists based on the static score and scanning the entire inverted lists (i.e. no "early-termination"). For our experiments we used the same scoring method as the one employed by Trevi. The Trevi scoring method uses a non-trivial combination of a TF.IDF.-based dynamic score and an in-degree static score. For the experiments with PageRank static score, we produced both the experimental results and the ground-truth results using a modified version of the Trevi scoring method that uses PageRank instead of in-degree.

We present the differences between the results returned with static score bucketing and the ground-truth results using the Kendall Tau similarity measure [8]. Intuitively, this measure counts the number of inversions in ranks for each pair of elements of both ranked lists and penalizes each inversion. We also performed experiments using the Spearman Rho and Spearman Footrule measures. These experiments showed similar results to the ones using the Kendall Tau measure and are not presented.

To better characterize the quality of the produced results, we used the fol-

lowing parameters specific only to this set of experiments.

The *TermThreshold* parameter controls when the scanning of the inverted lists is terminated. There has been little research on methods for early termination for inverted lists with static-score or document-id order (see [22]). Previous research for early termination in inverted lists with term-frequency order has focused on the case of TF-IDF scoring. In this case, there is a significant correlation between the position of the document in the inverted list, its score, and the query evaluation progress. Therefore, the stopping criteria are based on the score of the currently- processed document. In our case, this is not applicable because there is much less correlation between the position of the document and its final score due to the use of both static and dynamic score. Therefore, for our experiments, the stopping criteria is purely based on the position of the document in the inverted list: the *TermThreshold* parameter is the number of document postings scanned. Lower values model query processing under heavy load. In such cases, the query engine cannot allocate a lot of resources (CPU time, memory, etc.) per query. Higher values model query processing under less stress when the query engine can process most or all of the information. Unlike the other possible stopping criterion, the query evaluation time, the use of number of scanned documents allows for repeatable experiments.

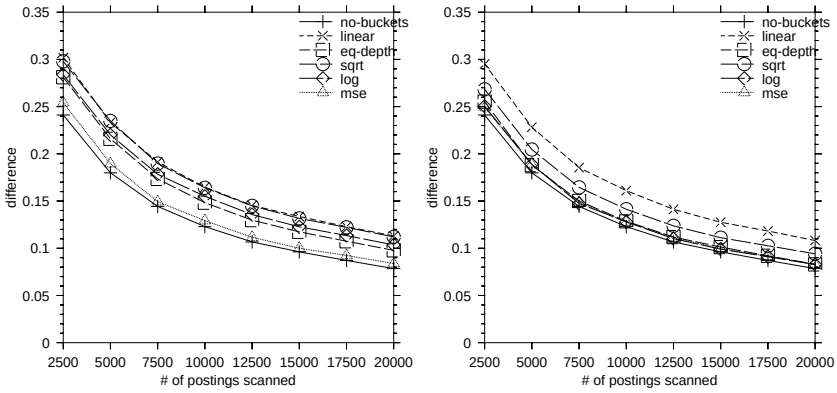
The *TopK* parameter specifies how many of the top results are considered when comparing the ranked lists. The default value is set 200 to represent the overall quality of the results. We also use values of 5 and 10 to study the quality of the top few results, which are most often considered by the users.

The *Ties* parameter controls the use of ties when comparing the ranked lists. When ties were used all documents with the same score are assigned the same rank. This is unlike the default case of comparison with no ties where the arbitrary ordering of documents with the same score leads to differences in the rank. Thus, the default case represents the typical situation where the query algorithm always enforces certain ordering and this ordering interacts with set of returned results. Therefore, the difference without ties shows the "observed" (by the user) difference. The comparison with ties represents the "true" difference of the ranked lists.

Figures 5(a), 5(b) present the experiments using *StaticScore* = in-degree and *BucketNum* = 4 and 64 respectively, and no ties. These figures compare 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 in the results is significant when a small portion of the inverted lists is scanned. It quickly decreases when the engine scans larger portions of the index. It drops almost twice when changing from 2500 scanned postings (3.1% of the average number per inverted list) to 17500 scanned postings (22%). 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. This is an indication that bucketing is indeed a viable solution.

Furthermore, all three sets of experiments show that the differences between



(a) BucketNum=4 (b) BucketNum=64

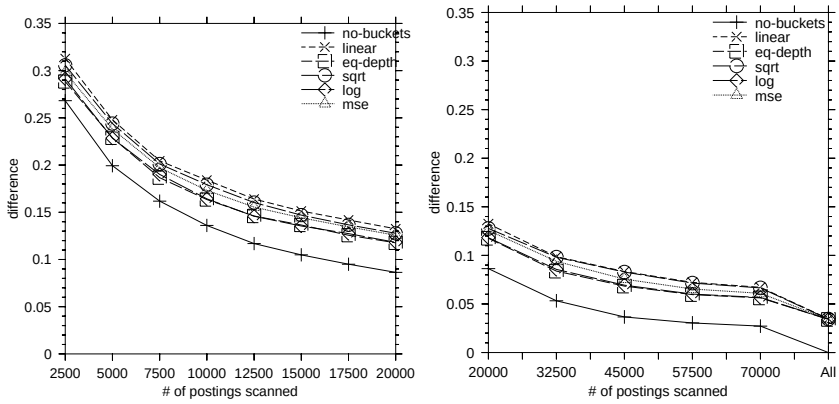
Figure 5: StaticScore=in-degree

the various bucketing types are small. The only exception is the uniform (linear) bucketing which is consistently worse from the others. This is not surprising since given the power-law distribution of in-degree. Almost all postings end up in one bucket while the other buckets have a very small number of postings. On the other hand, the query engine scans a small portion of the last bucket. This increases the probability of missing important results, which in turn degrades the quality of search results. The sub-optimal behavior of linear bucketing can also be observed from the lack of a trend for improvement when the number of buckets increases.

On the other end of the spectrum, the exponential ("exp") bucketing performs slightly better than the others. This can be attributed to the fact that it adapts the bucketing to the power-law distribution of the data. Thus, the buckets are more meaningful clustering of the documents in terms of their static score. As it can be seen, even for 4 buckets, the exponential bucketing is very close to "no-buckets". This leaves room for only a slight improvement when the number of buckets is increased.

All other bucketing schemes fall in-between "linear" and "exp" in terms of result quality. They also show significant improvements when increasing the number of buckets. When $BucketNum = 64$, almost all of them are indistinguishable from "exp". Even though, "exp" seems to perform consistently better, it has the additional cost for computing the power parameter of the power-law distribution. All other bucketing schemes do not need such additional computations and thus have the advantage of less computationally intensive implementation.

The experiments with *StaticScore*= PageRank show similar trends although they also exhibited some interesting differences. Due to the lack of space, we comment only on the results when $BucketNum = 64$, shown on Figures 6(a) and



(a) TermThreshold $\leq$ 20000 postings scanned

(b) TermThreshold $\geq$ 20000 postings scanned

Figure 6: StaticScore=PageRank, BucketNum=64

6(b). The results for $BucketNum = 4$ and $BucketNum = 16$ support the same conclusions.

Figures 6(a) and 6(b) show the difference to the base results for both lower and higher values of the *TermThreshold* parameter. As in the case of in-degree, there are minor differences between the different methods. There is a slight change for "linear", which is not significantly worse than the others. We attribute that to the fact that the higher precision of the PageRank scores (doubles as compared to integers for in-degree) allows for better clustering of the postings into buckets. In the case of in-degree, most of the documents have score 1 or 2, which either makes either no difference or a huge difference. Overall, when using PageRank, it is even less important which type of bucketing is used.

In contrast to the case with in-degree, all the bucketing methods seem to perform noticeably worse than the reference method "no-bucket". Again, this can be attributed to the better precision of PageRank. In the case of "no-bucket" with in-degree, there is still a large amount of randomness in the decision on how all the documents with in-degree 1 or 2 are stored in the inverted list. Even though all these documents are technically sorted, the coarser score granularity acts as a bucketing method: there is no difference among all documents with score 1.

Finally, it should be noted that the presence of a difference between "no-bucket" and the rest of the methods, even when the entire inverted lists are scanned, is due to the similarity measure and the use of top-k evaluation. The search engine implementation uses a heap to keep the current top-k candidates. For efficiency reasons, a document is inserted in the heap only if it has score

that is larger than smallest score in the heap. Thus, it is important for the documents with the lowest scores in the heap, which document was scanned first. Bucketing can introduce such small variations. On the other hand, we already noted that we find this artifact important because the user is always presented some total order and we should account for such side effects.

The experiments with ties (Figure 7(a)) showed interesting results. The quality of most of the algorithms (except "linear") increases by up to 30%. Therefore, the quality of the produced results is actually better than the experiments without ties show.

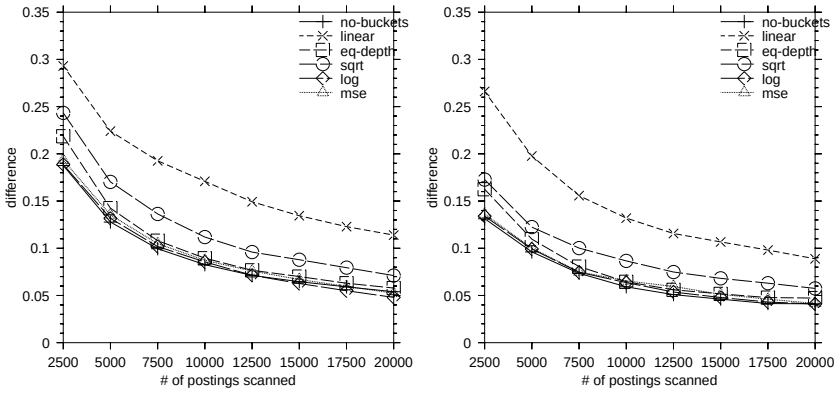
The experiments with lower values for $TopK$ also show interesting results. In Figure 7(b), we present only the results for $TopK = 5$. The results for $TopK = 10$ are similar. "Linear" is again an outlier and there is almost no change. We can see that the difference to the ground truth decreases almost by half for most of the other bucketing methods. This improved difference shows that the bucketing is able to preserve the top results, which are usually the most important ones.

In summary, the experiments on result quality show that good result quality can be achieved for even a small number of buckets. This is important because the smaller number of buckets is generally good for the indexing algorithm. Due to the coarser quantization, less documents will change their static score bucket when their static score changes. Furthermore, the experiments show that the bucketing method does not significantly influence the result quality (with the possible exception of "linear"). Other factors may outweigh the benefits in result quality. For example, "exp" will probably perform well only when it fits the static score distribution. As another example consider the "eq-depth" bucketing. In general, it generates good results but may has an additional overhead at indexing time: we need to know all the scores to determine the buckets. Finally, the agreement in the results for the two static scores tested, PageRank and in-degree, suggest that the results may hold for a larger variety of static scoring methods. We plan to experimentally establish this in our future work.

6 Related Work

Inverted list indexes [13] have been found to give best performance for information retrieval [24]. Thus, a lot of research has focused on methods for their efficient generation and maintenance. The initial research focused on indexing of static document collections. More recently, the focus shifted to the problem of updating inverted indexes in dynamic document collections [5, 6, 16, 19, 20]. Most of the research focuses on inverted lists in document order where insertions are done by appending postings to the inverted lists.

Our work is related to the research on early termination for query evaluation. Previous research has focused on early termination algorithms for score-ordered inverted lists [1, 17]. The inverted lists are ordered by term frequency and there is no study on the use of bucketing for improving index generation time. It



(a) Ties, BucketNum=64

(b) Top-5, BucketNum=64

Figure 7: StaticScore=in-degree

should be noted that Ngoc Anh [1] studies score bucketing and its influence on query results but its application is in inverted list compression. In this aspect, it is closer to Haveliwala's work [14]. The latter investigates bucketing to determine the optimal number of bits for static scores. Static score ordering for inverted lists is discussed by Long and Suel [22] and Fontoura [10]. However, none of these discusses the use of bucketing for improving index build performance and the influence of bucketing on query results.

There are hybrid organizations like XRANK [12] where the score order is augmented with additional B+-tree indexes that provide random access to the inverted list based on the document identifier.

7 Conclusions

In this paper, we presented an approach for inverted index organization that can be used to improve index build time. We proposed a relaxation of the total order of inverted list postings based on static score with a partial order: the inverted list is split into buckets and there is no constraint on the order within a bucket. We presented an algorithm for index generation that exploits this index organization to achieve a significant improvement in build time. We experimentally studied a class of methods for bucketing and showed that the gains in terms of index generations time can be achieved at only slight degradation in result quality. Our results were found to hold with both PageRank and in-degree. Thus, they are applicable for a large class of search engines. As a future work, we plan to study the applicability and the methods for bucketing in on-line inverted index generation.

References

[1] V. N. Anh, O. de Kretser, and A. Moffat. Vector-space ranking with effective early termination. In *Proc. of the 24th Annual Int. ACM SIGIR Conf. on Research and Development in Information Retrieval*, pages 35–42. ACM Press, 2001.

[2] R. Baeza-Yates and B. Ribero-Neto. *Modern Information Retrieval*. Addison-Wesley, 1999.

[3] 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.

[4] A. Z. Broder, D. Carmel, M. Herscovici, A. Soffer, and J. Y. Zien. Efficient query evaluation using a two-level retrieval process. In *Proceedings of the 2003 CIKM International Conference on Information and Knowledge Management*, pages 426–434, 2003.

[5] C. Clarke, G. Cormack, and F. Burkowski. Fast inverted indexes with online update. In *Technical Report CS-94-40, University of Waterloo*, 1994.

[6] D. Cutting and J. Pedersen. Optimizations for dynamic inverted index maintenance. In *Proc. of the 13th Int. ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 405–411, 1990.

[7] R. Fagin, R. Kumar, K. McCurley, J. Novak, D. Sivakumar, J. Tomlin, and D. Williamson. Searching the workplace web. In *Proc. of the 12th International World Wide Web Conference*, 2003.

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

[9] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. In *Symposium on Principles of Database Systems*, 2001.

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

[11] R. Gray and D. Neuhoff. Quantization. *IEEE Transactions on Information Theory*, 44(6):2325–2383, 1998.

[12] L. Guo, F. Shao, C. Botev, and J. Shanmugasundaram. Xrank: Ranked keyword search over XML documents. In *SIGMOD'2003*, 2003.

[13] D. Harman, E. A. Fox, R. A. Baeza-Yates, and W. C. Lee. Inverted files. In *Information Retrieval: Data Structures & Algorithms*, pages 28–43. 1992.

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

[15] 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*.

[16] L. Lim, M. Wang, S. Padmanabhan, J. Vitter, and R. Agarwal. Dynamic maintenance of web indexes using landmarks. In *Proc. of WWW Conference 2003*, 2003.

[17] M. Persin, J. Zobel, and R. Sacks-Davis. Filtered document retrieval with frequency-sorted indexes. *J. Am. Soc. Inf. Sci.*, 47(10):749–764, 1996.

[18] G. Salton and C. Buckley. Term-weighting approaches in automatic text retrieval. *Inf. Process. Manage.*, 24(5):513–523, 1988.

[19] K. Shoens, A. Tomasic, and H. Garcia-Molina. Synthetic workload performance analysis of incremental updates. In *Proc. of the 17th Annual Int. ACM SIGIR Conf. on Research and Development in Information Retrieval*, pages 329–338. Springer-Verlag New York, Inc., 1994.

[20] A. Tomasic, H. Garcia-Molina, and K. Shoens. Incremental updates of inverted lists for text document retrieval. In *1994 ACM SIGMOD Int. Conf. on Management of data*, pages 289–300. ACM Press, 1994.

[21] D. H. W. Kraaij, T. Westerveld. The importance of prior probabilities for entry page search. In *SIGIR 2002*, 2002.

[22] 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.

[23] A. X. Zheng, A. Y. Ng, and M. I. Jordan. Stable algorithms for link analysis. In *Proceedings of 24th ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR '01)*, pages 258–266, New Orleans, Louisiana, USA, September 2001.

[24] J. Zobel, A. Moffat, and K. Ramamohanarao. Inverted files versus signature files for text indexing. *ACM Trans. Database Syst.*, 23(4):453–490, 1998.