Improving the Performance of Pipelined Query Processing with Skipping

Simon Jonassen and Svein Erik Bratsberg

Norwegian University of Science and Technology, Trondheim, Norway,
(simonj,sveinbra)@idi.ntnu.no

Abstract. Web search engines need to provide high throughput and short query latency. Recent results show that pipelined query processing over a term-wise partitioned inverted index may have superior throughput. However, the query processing latency and scalability with respect to the collections size are the main challenges associated with this method. In this paper, we evaluate the effect of inverted index skipping on the performance of pipelined query processing. Further, we introduce a novel idea of using Max-Score pruning within pipelined query processing and a new term assignment heuristic, partitioning by Max-Score. Our current results indicate a significant improvement over the state-of-the-art approach and lead to several further optimizations, which include dynamic load balancing, intra-query concurrent processing and a hybrid combination between pipelined and non-pipelined execution.

1 Introduction

Two fundamental index partitioning methods, *term-wise* and *document-wise*, have been extensively compared during the last 20 years. The decision about which method gives the best performance relies on a number of system aspects such as query processing model, collection size, pruning techniques, query load, network and disk-access characteristics [3]. Rather than asking whether term-wise partitioning is generally more efficient than document-wise or not, we look at *pipelined query processing* [10] over a term-wise partitioned index and try to resolve the greatest challenges of the method. The advantage of term-wise partitioning is that a query containing l query terms involves only l processing nodes in the worst case. At the same time, it has to process only l posting lists in total. Whether l is the number of lexicon lookups or the total number of disk-accesses depends on the implementation. Pipelined query processing consists of routing a query bundle through the nodes responsible for any of the terms appearing in a particular query, modifying the accumulator set at each of these nodes and finally extracting the results at the last node. Compared to a traditional query processing approach, where each of the nodes retrieves the posting lists and the query processing itself is done solely by a ranker node, this method improves the workload distribution across the nodes and the overall performance [10].

In a recent book, Büttcher et al. [1] enlist three problems of pipelined query processing: poor scalability with increasing collections size, poor load balancing

and limited intra-query concurrency due to term-at-a-time/node-at-a-time processing. In this work, we mainly address the first problem, but also show how our solution can be used to solve the other two.

Our contributions in this work are as follows. First, we explore the idea of combining inverted index skipping and pipelined query processing for a term-wise partitioned inverted index. Herein, we present an efficient framework and a skipping optimization to the state-of-the-art approach. Second, we present a novel combination of pipelined query processing, Max-Score pruning and document-at-a-time sub-query processing. Third, we present an alternative posting-list assignment strategy, which improves the efficiency of the Max-Score method. Fourth, we evaluate our methods with a real implementation using the TREC GOV2 document collection, two large query sets and 9 processing nodes and suggest a number of further optimizations.

The remainder of this paper is organized as follows. We briefly review the related work in Section 2 and describe our framework in Section 3. We present our query processing optimizations in Sections 4 and 5 and our posting list assignment optimization in Section 6. We summarize our experiments in Section 7. In Section 8, we finally conclude the paper and suggest further improvements.

2 Related work

Pipelined query processing was originally presented by Moffat et al. [10] as a method that retains the advantage of term-wise partitioning and additionally reduces the overhead at the ranker node. However, due to a high load imbalance, the method was shown to be less efficient and less scalable than document-wise partitioning. In the following work by Moffat et al. [9] and Webber [13], the load balancing was improved by assigning posting lists in a fill-smallest fashion according to the workload (posting list size $\times$ past query frequency) associated with each term. The authors suggested also to multi-replicate the posting lists with the highest workloads and allow the broker to chose the replica dependent on the load. Additionally, Webber [13] has presented a strategy where each node may ask other nodes to report their current load in order to choose the next node in the route. The results reported by Moffat et al. [9] showed a significant improvement over the original (term-wise) approach, but not the document-wise approach. Under extended evaluation with a reduced main memory size performed by Webber [13] pipelined query processing outperformed document-wise partitioning in terms of maximum throughput. However, document-wise partitioning demonstrated shorter latency at low multiprogramming levels.

In a recent publication [4], we have addressed the problem of the increased query latency due to a strict node-at-a-time execution and presented a semi-pipelined approach, which combines parallel disk access and decompression and pipelined evaluation. Additionally, we suggested to switch between semi- and non-pipelined processing dependent on the estimated network cost and an alternative query routing strategy. While the results of this work indicate an ultimate trade-off between the two methods, it has several important limitations. First, similar to the original description [13], it requires to fetch and decompress post-

ing lists completely. This minimizes the number of disk-accesses, but leaves a large memory footprint and endangers scalability of the method. Second, while the original approach fetches only one of a query's posting lists at a time, the semi-pipelined approach fetches all of them and keeps them in main memory until the query has been processed up-to and including this node. This improves the query latency, but increases the memory usage even further. Third, the compression methods used by the index may be considered outdated.

In the current work, we follow an alternative direction and try to improve pipelined query processing by means of skipping optimizations. In a recent work [5], we have investigated these techniques for a non-distributed index. In particular, we have presented a self-skipping inverted index designed specifically for NewPFoR [14]. Further, we have presented a complete description of document-at-a-time Max-Score. The Max-Score heuristic was originally presented by Turtle and Flood [12] without specifying enough details for skipping itself and later a very different description was given by Strohman et al. [11]. Our algorithm combines the advantage of both previous descriptions. Finally, we have presented a skipping version of the space-limited pruning algorithm by Lester et al. [7], which is the method used in the original description of pipelined query processing.

As a part of the current work, in Section 8, we outline several further extensions. One of these, intra-query concurrent pipelined processing, has already been evaluated in our recent work [6]. The preliminary baseline of [6] is an early version of the Max-Score optimization we are about to present, $\mathrm{MSD}^{\star}_{\mathrm{s}}$, and the results of [6] are therefore directly applicable to this method.

In contrast to the previous work, we use skipping to resolve the limitations of distributed, pipelined query processing. Herein, we present several skipping optimizations and a new term assignment strategy. In contrast to the previously presented assignment optimizations [2,8,15], our strategy does not try to assign co-occurring terms to the same node or to do load balancing, but rather to maximize the pruning efficiency. Additionally, it opens a possibility for dynamic load balancing with low repartitioning overhead and hybrid query processing. Finally, different from [4,5,6], our experiments use two query logs with very different characteristics, and a varied collection size.

3 Preliminaries

For a given textual query q, we look at the problem of finding the k best documents according to a similarity score $sim(d, q) = \sum_{t \in q} s(t, d)$, where s is a function, such as Okapi BM25, estimating the relevancy of a document d to a term t. For this reason, we look at document-ordered inverted lists. Each list I_t stores an ordered sequence of document IDs where the term t appears and an associated number of occurrences of the term in the document $f_{t,d}$.

Our search engine runs on $n+1$ processing nodes, one of which works as a query broker and the remaining n work as query processing nodes. For each of the indexed collection terms we build an inverted list and assign it to one of the processing nodes. By default, we use a hash-based term assignment strategy. Additionally to the inverted index, each query processing node maintains a

small lexicon (storing term IDs, inverted file pointers, document and collection frequencies), a small replica of the document dictionary (storing the number of tokens contained in each document) and both partition and collections statistics. The query broker node stores a full document dictionary (document IDs, names and lengths), lexicon (tokens, term IDs, collection and document frequencies, IDs of the nodes a term is assigned to) and collection statistics. At runtime, the inverted index resides on disk and the remaining structures are in main memory.

The broker is responsible for receiving queries, doing all necessary preprocessing and issuing query bundles. The broker uses its lexicon to look-up query terms and to partition the query into several sub-queries consisting of the terms assigned to the same node. Further, it calculates a query route (i.e., a particular order to visit the query nodes) and creates a query bundle. The bundle contains term IDs, query frequencies and an initially empty document-ordered accumulator set. Accumulators represent document IDs and partial scores. Finally, the broker sends the query bundle to the first node in the route.

When a node receives a bundle, it decompresses its content and starts a new query processing task. First, it uses its own lexicon to find the placement of the posting list. In the next step, the node processes its own posting data with respect to the received accumulators and creates a new accumulator set. Then, it updates the query bundle with the new accumulator set and transfers it to the next node. Alternatively, the last node in the route extracts the top-k results, sorts them by descending score and returns them to the broker.

4 Improving pipelined query processing with skipping

In this section, we describe the optimization to the state-of-the-art query processing approach. With this approach, the terms within each query are ordered by increasing collection frequency F_t and the query itself is routed by increasing minimum F_t. Once a bundle is received by a node, the node extracts the accumulators and initiates posting list iterators. We use Algorithm 1 to describe the following processing step performed by the query processor. The space-limited pruning method itself has been originally presented by Lester et al. [7] and the skipping optimization for a non-distributed index has already been presented in our previous work [5]. Therefore, the goal of this section is to describe the improvements to pipelined query processing. However, for the sake of intelligibility we also explain the most important details derived from the prior work [5,7].

In the following, i_t denotes the iterator of the posting list I_t, which provides methods to get the document ID, frequency, score and position of the current posting, advance to the next posting or to the first posting having $d \geq d'$ (both $next()$ and $skipTo(d')$ return *false* if the end has been reached), or reset to the beginning. Further, f_t denotes the document frequency of t (i.e., number of documents) and F_t - its collection frequency (i.e., number of occurrences), and $|A|$ - the current size of the accumulator set A.

The idea behind the pruning method (lines 1-7, 9, 11) is to restrict the accumulator set to a target size L. As the algorithm shows, the posting lists of

Algorithm 1: processBundle(b_q) with skip-optimized space-limited pruning

Data: iterators $\{i_t\}$ and lexicon entries $\{l_t\}$ for $t \in q_j$ sorted by ascending F_t, accumulator set A, bundle attribute v, system parameters *avg_dl*, *use_opt*, L, $h_{\max}$

```
1  foreach t ∈ q_j do
2      A′ ← A, A ← ∅, skipmode ← false, p ← ⌈f_t/L⌉;
3      if f_t < L then
4          p ← f_t + 1, h ← 0;
5      else if v = 0 then
6          set h to the maximum of the first p frequencies retrieved from i_t,
           v ← s(l_t, h, avg_dl), i_t.reset();
7      else
8          if use_opt = true and s(l_t, h_max, avg_dl) < v then skipmode ← true;
9          else find h ∈ [1, h_max] s.t. s(l_t, h, avg_dl) ≥ v;
10     if skipmode = false then
11         s ← max(1, ⌊(h+1)/2⌋), size_0 ← |A′|; merge A′ and candidates from i_t into A:
           calculate i_t.s() only when i_t.d() ∈ A′ or i_t.f() ≥ h, prune candidates having s < v;
           Each time i_t.pos() = p: pred ← |A| + |A′| + (f_t − p) × (|A| + |A′| − size_0)/p, if
           pred > 1.2 × L then h ← h + s else if pred < L/1.2 then h ← max(0, h − s) endif,
           v ← s(l_t, h, avg_dl), s ← ⌊(s+1)/2⌋, p ← 2 × p;
12     else
13         foreach accumulator ⟨d′, s⟩ ∈ A′ do
14             if i_t.d() < d′ then if i_t.skipTo(d′) = false then add remaining accumulators
               s.t. s ≥ v to A, proceed to the next term (line 1);
15             if i_t.d() = d′ then s ← s + i_t.s();
16             if s ≥ v then add ⟨d′, s⟩ to A;
17 if it is the last node in the route then use a min-heap to find the k-best candidates from A,
   sort and return them to the broker else update b_q with A and v and send it to the next node;
```

a particular sub-query $q_j \subseteq q$ are evaluated term-at-a-time. As long the document frequency f_t of the current term is below L, each posting is scored and merged with the existing accumulator set. Otherwise, the algorithm estimates a frequency threshold h and a score threshold v, just as suggested by Lester et al. For this reason, h is set to the maximum frequency among the first $p = \lceil f_t/L \rceil$ postings. The rationale here is that, if one out of p postings will pass the frequency filter $f_{t,d} \geq h$, the total number of such postings will be L. Further, in order to prune existing accumulators, v is calculated from h. Differently from Lester et al., our score computation uses also the length of a document. For this reason we calculate the threshold score using l_t, h and the average document length *avg_dl*. Now, the algorithm is able to prune the existing accumulators having score $s < v$ and avoid scoring postings having $f_{t,d} < h$ and not matching in among the existing accumulators. Each time p postings of I_t has been processed, it predicts the size of the resulting accumulator set. Then, it either increases or decreases the frequency threshold, updates the score threshold and finally cools-down the threshold variation. Finally, if the thresholds have already been defined, it uses the previously computed v to find the corresponding value h. Similar to the implementation in Zettair[1], we try only the values between 1 and a system-specific maximum value $h_{\max}$. Additionally, we apply binary search to reduce the number of score computations.

[1] http://www.seg.rmit.edu.au/zettair/

Our optimization (lines 8, 10, 12-16) suggests to switch the query processing into a conjunctive skip-mode when $s(l_t, h_{\max}, avg_dl)$ is below the previously computed v. In this mode, a posting is scored only when there is already an existing accumulator with the same document ID. For the first sub-query, b_q is initiated with $v = 0$. After processing a sub-query q_j, b_q is updated with the current A and v and forwarded to the next node. This means that each query starts in the normal, disjunctive mode. When the optimization constraint holds, it switches into the conjunctive mode. However, if the next posting list is shorter than L, the processing will switch back to the normal mode, but it will proceed to prune the accumulators having $s < v$.

The benefit of our optimization depends also on the inverted index implementation. We apply the layout described in our previous work [5]. With the basic index, each posting list is divided into groups of 128 entries, which are stored as two chunks containing 128 document ID deltas and 128 frequencies, both compressed with NewPFoR. To support skipping, we build a hierarchy of skip-pointers, which consist of an end-document ID and an end-offset pointer to a chunk level below. Further, we calculate deltas and compress these in chunks of 128 entries using NewPFoR. Next, we prefix-traverse the logical tree while writing to disk in order to minimize the size of skip-pointers and optimize reading. At the processing time, each posting list iterator maintains one chunk from each skip-level decompressed in main memory and applies buffering while reading from disk. Different from the previous work, we decompress frequency chunks only when at least one of the corresponding frequencies has been requested.

With the basic index, the cost of a skip is proportional to the number of blocks (I/O) and chunks (decompression) between the two positions. With the self-skipping index, the operation is done climbing the logical hierarchy up (using already decompressed data) and down (reading and decompressing new data when necessary). Therefore, the upper bound cost of the operation in the number of decompressed chunks and read blocks is $O(\log(D))$.

5 Max-Score optimization of pipelined query processing

The main drawback of the query processing methods discussed in the previous section lies in the unsafe pruning strategy. Additionally, these techniques are limited to term-at-a-time and node-at-a-time query processing. For this reason, we suggest an alternative query processing approach employing document-at-a-time processing of sub-queries, and later we show how this new method can be extended to provide intra-query parallelism. In order to guarantee safe pruning, we look at the Max-Score heuristic [5,11,12]. To give a better explanation, first we describe the idea for a non-distributed scenario, $q_j = q$, then how it can be applied to pipelined query processing, and finally, present the algorithm.

At indexing time, we pre-compute an upper-bound score $\hat{s}_t$ for each posting list I_t (i.e., the maximum score that can be achieved by any posting). At query processing time, we order $q_j = \{t_1, \ldots, t_l\}$ by decreasing $\hat{s}_t$ and use a_i to denote the maximum score of terms $\{t_i, \ldots, t_l\}$, i.e., $a_i = \sum \hat{s}_t$ s.t $t \in \{t_i, \ldots, t_l\}$. Further, q_j is processed document-at-a-time and each iteration of the algorithm

selects a new candidate, accumulates its score and eventually inserts it into a k-entry min-heap. The idea behind Max-Score is to prune the candidates that cannot enter the heap. As terms are always processed in order t_1 to t_l, they can be viewed as two subsets $\{t_1, \ldots, t_r\}$ (required) and $\{t_{r+1}, \ldots, t_l\}$ (optional), where r is the smallest integer such that $a_r \geq \tilde{s}$ and $\tilde{s}$ is the current k-th best score. It is easy to see that candidates that do not match any of the required terms cannot enter the heap. Therefore, the candidate selection can be based only on the required terms, which also have shorter posting lists. Once a candidate is selected, the terms are evaluated in order and the optional term iterators are advanced with a skip. Finally, at any point, a partially scored candidate can be pruned if its partial score plus the maximum remaining contribution is below the score of the current k-th best candidate, i.e., $s+a_i<\tilde{s}$. The description is so far similar to [5].

Now we explain how to apply these ideas to pipelined query processing. At query processing time, the broker fetches maximum scores along with term ID and location information and includes them in the query bundle. Therefore, when b_q arrives at a particular node, the information about the maximum scores of the terms in the current sub-query, $\hat{s}_t$ s.t. $t \in q_j$, and the maximum contribution of the remaining sub-queries, $\tilde{a} = \sum \hat{s}_t$ s.t. $t \in q_i$, $q_i \subseteq q$ and $i > j$, are available to the query processor. b_q itself is routed by decreasing maximum $\hat{s}_t$ among the sub-queries. Each query processor treats the received accumulator set just as a posting list iterator with $\hat{s}_t$ set to highest score within the set, and processes the sub-query document-at-a-time. Therefore, the required subset is defined by $a_r \geq \tilde{s} - \tilde{a}$, any candidate can be pruned whenever $s+a_i<\tilde{s}-\tilde{a}$ holds, and finally, the candidates with partial scores $s \geq \tilde{s}-\tilde{a}$ have to be transferred to the next node as a modified accumulator set.

We use Algorithm 2 to describe the final query processing approach. First, the algorithm prepares for query processing, calculates a values and defines the required set (lines 1-4). As long as the required set is non-empty, the iterators are processed document-at-a-time (lines 5-13). Each iteration advances the recently used iterators of the required set, selects a new candidate (lines 6-7) and accumulates its score (lines 8-11). If an iterator reaches the end of the posting list, it is removed from the iterator set and the a values of remaining iterators and r are updated (lines 6 and 10). If a candidate succeeds to reach a score higher than the pruning threshold ($s \geq \tilde{s}-\tilde{a}$), it is inserted in the accumulator set (line 12). Potentially, it may also be inserted into the candidate heap (line 13). In this case, $\tilde{s}$ may also be updated. When a sub-query is fully processed, if this is the last node in the route, the candidate heap has to be sorted and returned to the broker as the final result set (lines 14-15). Otherwise, non-pruned accumulators have to be transferred to the next node. In this case, prior to the final transfer, an extra pass through the accumulator set removes candidates having $s<\tilde{s}-\tilde{a}$, which are false positives due to a monotonically increasing pruning threshold. In order to facilitate pruning, $\tilde{s}$ is initiated with the value received from the previous node, or 0 for the first node. In practice, the last node in the route does not have to store non-pruned accumulators, but only the candidate heap.

Algorithm 2: processBundle(b_q) with DAAT Max-Score optimization

Data: iterators $\{i_1, \ldots, i_l\}$ and maximum scores $\{\hat{s}_1, \ldots, \hat{s}_l\}$ sorted by descending $\hat{s}$, accumulator set A, bundle attributes $\tilde{s}$ and $\tilde{a}$

1 **if** $A \neq \emptyset$ **then** $l \leftarrow l+1$, **for** $x \leftarrow l$ *to* 1 **do** $i_x \leftarrow i_{x-1}, \hat{s}_x \leftarrow \hat{s}_{x-1}$ **endfor**, set i_1 to be A's iterator and $\hat{s}_1$ to be A's maximum score;
2 $a_l \leftarrow \hat{s}_l$, **for** $x \leftarrow l-1$ *to* 1 **do** $a_x \leftarrow a_{x+1}+s_x$ **endfor**, $A' \leftarrow \emptyset$, $minHeap \leftarrow \emptyset$;
3 $r \leftarrow l$, **while** $r>0$ *and* $a_r < \tilde{s}-\tilde{a}$ **do** $r \leftarrow r-1$;
4 $d' \leftarrow -1$;
5 **while** $r>0$ **do**
6   advance iterators having $i_{x \leq r}.d() = d'$, if $i_x.next() = false$: close i_x, update i, s and a sets, decrement l, recompute r (similar to line 3, break if $r=0$);
7     $d' \leftarrow \min(i_{x \leq r}.d())$, $s \leftarrow 0$;
8   **for** $x \leftarrow 1$ *to* l **do**
9   **if** $s+a_x < \tilde{s}-\tilde{a}$ **then** break and proceed to selection of the next candidate (line 5);
10   **if** $x>r$ *and* $i_x.d() < d'$ **then** advance i_x to d', if $i_x.skipTo(d') = false$: close i_x, update i, s, a, l and r (break if $r=0$) and proceed to the next iterator (line 8);
11   **if** $i_x.d() = d'$ **then** $s \leftarrow s+i_x.\mathrm{s}()$;
12   **if** $s \geq \tilde{s}-\tilde{a}$ **then** add $\langle d', s\rangle$ to A';
13   **if** $s > minHeap.minScore$ **then** add $\langle d', s\rangle$ to $minHeap$, $\tilde{s} \leftarrow \max(\tilde{s}, s)$;
14 **if** it is the last node in the route **then**
15   retrieve candidates from $minHeap$, sort and return them to the broker;
16 **else** $A \leftarrow \{\langle d, s\rangle \in A'$ s.t. $s \geq \tilde{s}-\tilde{a}\}$, update b_q with A and $\tilde{s}$ and send it to the next node;

6 Max-Score optimization of term assignment

The pruning performance of the Max-Score optimization can be limited when long posting lists appear early in the pipeline. Therefore, we suggest to assign posting lists by decreasing $\hat{s}_t$, such that the first node gets posting lists with highest $\hat{s}_t$ and the last node gets posting lists with lowest $\hat{s}_t$. For simplicity, we use equally sized partitions. Since $\hat{s}_t$ increases as f_t decreases (because most of the similarity functions, including BM25, use the inverse document frequency), this strategy implies that the first node now stores only short posting lists and the last node stores a mix of long and short posting lists, and the nodes in between store posting lists with short-to-moderate lengths.

As we show in the next section, this technique significantly improves the performance, but struggles with a high load imbalance. Beyond the experiments presented in the next section, we have tried several load balancing approaches similar to Moffat et al. [9], such as estimating the workload associated with each term t in a past query log Q', $L_t = |I_t| \times f_{t,Q'}$, where $f_{t,Q'}$ is the frequency of t in Q' and $|I_t|$ is the size of I_t, and splitting the index so the accumulated past load would be balanced across the nodes. However, since the load estimator does not take skipping into account, it overestimates the load of long posting lists and assigns nearly half of the index to the first node. As a result, the load imbalance gets only worse. Therefore, we leave load balancing as an important direction for further work and outline a possible solution in Section 8.

7 Experimental results

For our experiments, we index the 426GB TREC GOV2 corpus. With stemming and stop-word removal applied, it contains 15.4 mil. unique terms, 25.2 mil.

Table 1: Impact of the query processing method on the precision and recall of the Adhoc Retrieaval Topics 701-850.

Method	MAP	P@10	Recall
Full/MSD	.153903	.530872	.274597
LT1M	.153746	.530872	.274262
LT100K	.152376	.528859	.270955†
LT50K	.152378	.530872	.271049
LT25K	.150602†	.528188	.267361†
LT10K	.145877‡	.516107†	.256386$^{\sharp}$
AND	.155073	.531544	.268126

Table 2: Maximum and average document frequency and sample covariance between the document and query frequency distributions in the evaluated query sets.

Query set	$\max(f_t)$	$\text{avg}(f_t)$	$\text{cov}(f_t, f_{t,Q})$
A04-06	11256870	997968	149393
E05	11256870	223091	2266801
E06	11256870	290747	5999742

documents, 4.7 bil. pointers and 16.3 bil. tokens. With 8 index partitions, the resulting distributed index is 9.3GB in total, while a corresponding monolithic index is 7.6GB. Most of the overhead (1.54GB) comes from a short replicated version of the document dictionary. Skipping pointers increase the size by additional 87.1MB and the resulting index contains 279 647 posting lists with one skip level, 15 201 with two and 377 with three levels.

We run our experiments on a 9 node cluster. Each node has two 2.0GHz Quad-Core CPUs, 8GB memory and a SATA disk. The nodes are interconnected with a Gigabit network. Our framework[2] is implemented in Java. It uses Java NIO and Netty for efficient disk access and network transfer. For disk access, we use 16KB buffers and the default GNU/Linux OS caching policy (hence, we reset the disk caches before each run). Further, queries are preprocessed in the same way as the document collection and evaluated using the Okapi BM25 model.

We evaluate the following query processing methods. Full/non-pruned evaluation (Full), space-limited pruning described in Section 4 (LT denotes the state-of-the-art method, and SLT denotes the skip-optimized version), Max-Score optimized evaluation described in Section 5 (MSD), and finally an evaluation with intersection semantics and document-at-a-time sub-query processing (AND). We use a subscript N (e.g., SLT_N) to denote an execution on a non-optimized index and S on a self-skipping index. To limit the number of experiments, the maximum number of top-results k is fixed at 100. For LT we vary the accumulator set target size L, hence LT1M corresponds to $L = 1\ 000\ 000$ and LT10K to $L = 10\ 000$. Finally, we fix the $h_{\max}$ used by LT/SLT (see Sec. 4) at 2000, which we find to be suitable for our index.

In order to evaluate the impact of the query processing optimizations on the retrieval performance, we use the TREC Terabyte Track Adhoc Retrieval Topics and Relevance Judgments 701-850 from 2004, 2005 and 2006. We use documents with relevance judgments 1 and 2 as a ground truth and consider MAP, precision at 10 results (P@10) and recall at k results as retrieval performance indicators. Table 1 shows the averages over the whole query set. We focus on result degradation with the space-limited pruning (LT) compared to a full evaluation (Full),

[2] `https://github.com/s-j/laika`

which is the retrieval performance baseline. Beyond the average results, we apply a paired t-test at the query level to check the degree of significance. We use † to mark the significance at 0.05 level, ‡ at 0.01 and ♯ at 0.001.

Table 1 shows how the retrieval performance of LT degrades with decreasing L. Degradation becomes significant at lower values of L. The evaluation measures of LT50K and LT100K are different, but without statistical significance when compared to each other. Beyond the presented data, the results obtained with Full and LT10M are identical, and the results obtained with SLT are identical to LT for $L \geq 10\ 000$. From these results, we consider $L \geq 50\ 000$ as a suitable choice with respect to the precision and recall with $k = 100$ (while $L \leq 25\ 000$ is not) and keep LT100K, LT50K and LT25K for further experiments.

Our observations confirm that the results obtained by Full and MSD are identical. Furthermore, we observe a higher precision (MAP and P@10) with AND compared to Full, although the difference is statistically insignificant. A closer look has shown that AND performs better for topics 701-751 and 751-800 and worse for topics 801-850 (no significance). However, with $k = 1000$, Full has both a higher MAP (0.271470 versus 0.255441, no significance) and a higher recall (0.662522 versus 0.596165, significance at 0.01).

In order to evaluate the algorithmic performance, we use two subsets of TREC Terabyte Track Efficiency Topics from 2005 (E05) and 2006 (E06). Both subsets contain 20 000 queries that match at least one indexed term, where the first 5 000 queries are used for warm-up and the next 15 000 to measure the actual performance. To simulate the effect of a result cache, neither of the sets contains duplicated queries.

We observe that E06, which contains government-specific queries, implies higher query processing load than E05, which contains Web-specific queries. Therefore, we consider these two sets as a better (E05) and a worse case (E06) scenarios. We use Table 2 to illustrate the difference between the query sets, including the Adhoc topics marked as A04-06. As the table shows, the term with highest document frequency (i.e., posting list length) is the same in all three of the sets (which is the term 'state'), but the average document frequency and the sample covariance between the document frequency and the query set frequency are significantly different. A04-06 is a very short query set, with a flat query frequency distribution and missing a long-tail distribution among the document frequencies. This explains a high average document frequency and a low covariance. Finally, the results for E05 and E06 illustrate the load difference between these two sets. E06 contains terms with both longer posting lists and a higher correlation between the query and document frequencies.

Figure 1 illustrates the average latency per query (milliseconds) and overall throughput (queries per second) with varied multiprogramming levels. Points on each plot correspond to 1, 8, 16, 24, 32, 48, 56 and 64 concurrent queries (cq). Each run was repeated twice (with the disk cache being reset each time) and the average value was reported. We report results for all methods except Full, which was too slow – even when the multiprogramming level set to 1, a query took on average 474ms for E05, and 1121ms for E06. Therefore, we consider $\mathrm{LT_N}$

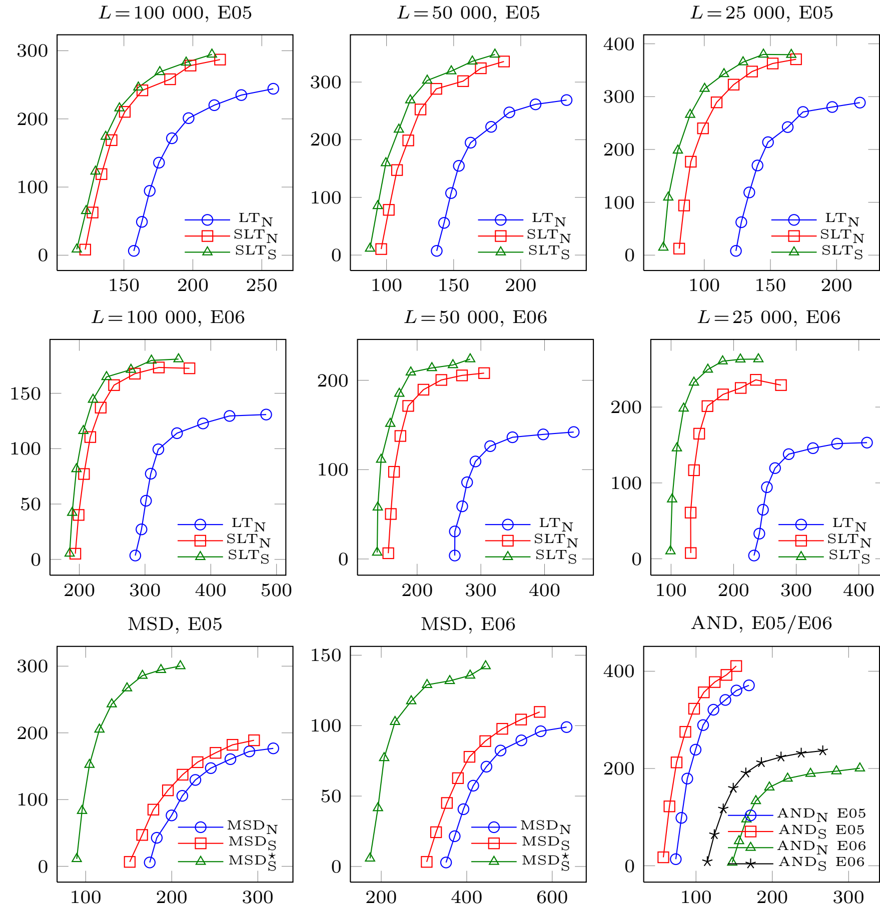

Fig. 1: Throughput (y-axis, qps) and latency (ms) with varied multiprogramming.

as the time performance baseline. Our results show that for both E05 and E06, the skipping optimization to LT (SLT_N) significantly improves the performance and the improvement increases with smaller L. Skipping support in the inverted index (SLT_S) provides a further improvement. While the index optimization is not as significant as the algorithmic optimization, we observe that (for E06) the improvement increases as L decreases. We explain these results using Figure 2, which illustrates the number of blocks read, chunks decompressed, unique document IDs evaluated, scores computed and accumulators sent and received by each node, normalized by the evaluation set query count. The figure shows both the average (across the nodes) and the maximum (one node) counts. As the results show, the improvement from Full_N to LT_N lies in reduced score computation and network transfer, which improve as L decreases. SLT_N further improves the number of candidates been considered (Doc.IDs) and SLT_S improves the amount of data been decompressed (Chunks). However, even with $L = 25\ 000$ there is no

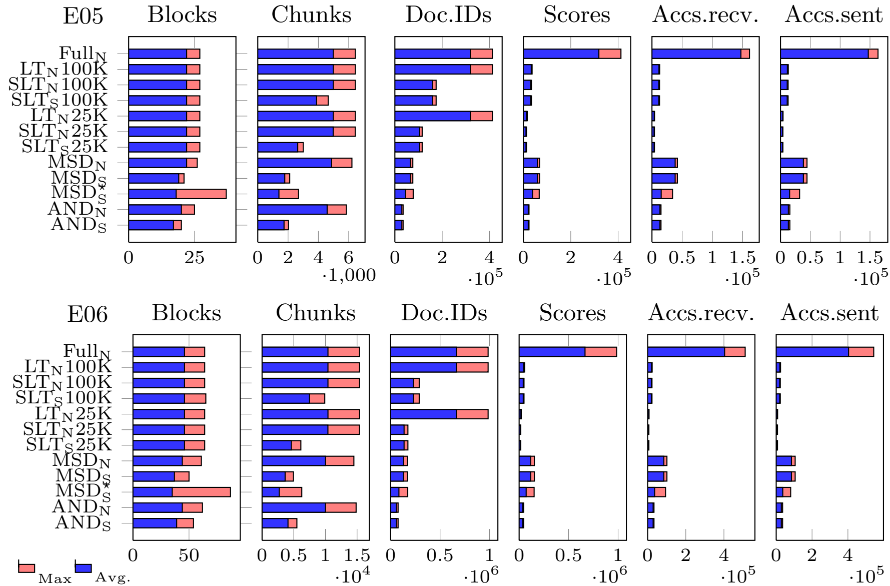

Fig. 2: Maximum and average number of processed entities per node.

additional savings in data read from disk (Blocks), which can be explained by a relatively large block size (16KB).

As we show in Figure 1, the Max-Score optimization (MSD$_{\mathrm{N}}$) gains a modest improvement from the self-skipping index (MSD$_{\mathrm{S}}$) and a significant improvement from the further term-assignment optimization (MSD$^{\star}_{\mathrm{N}}$). For E05, MSD$^{\star}_{\mathrm{N}}$ performs as good as SLT$_{\mathrm{S}}$100K, but for E06 it struggles with increasing query latency when compared to SLT$_{\mathrm{S}}$100K. However, having in mind that MSD is equivalent to a full (non-pruned) evaluation, it shows a very good performance. As Figure 2 shows, the Max-Score optimizations significantly improve the total amount of read, decompressed and evaluated data. While these methods increase the number of score computations and the number of transferred accumulators compared to the LT optimizations, they show a significant improvement compared to Full. Finally, our results show that the main challenge of MSD$^{\star}_{\mathrm{N}}$ is an increased load imbalance, which is the ratio between the maximum and the average counts. This issue should be investigated in the future.

As illustrated in Figures 1 and 2, skipping support in the index improves the performance of AND by reducing the amount of read, decompressed and evaluated data. For E06 it improves the latency at 1cq by 22% and the throughput at 64cq by 18%, for E05 it improves the latency at 1cq by 29% and the throughput at 64cq by 11%. Overall, AND performs better than SLT$_{\mathrm{S}}$25K on E05 and slightly worse on E06. As having a better result quality than LT25K on A701-850, we consider AND as a good alternative to the space-limited pruning with a low accumulator set target size (L).

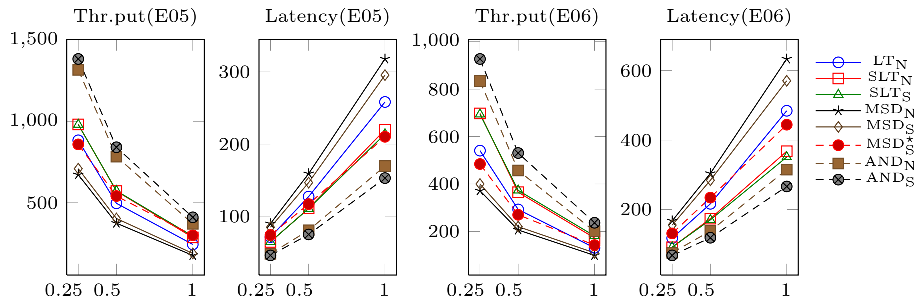

Fig. 3: Throughput (qps) and latency (ms) with varied collection size (x-axis).

Finally, we address performance linearity in Figure 3. In these results, we keep the multiprogramming level at 64cq and vary the collection size to 1/2 and 1/4. For LT/SLT we use $L = 100\ 000$ scaled with the collection size. The results show that with increasing index size, the methods converge in the absolute throughput and diverge in the latency. In our opinion, the best behavior is given by SLT, MSD and AND. However, our results do not guarantee the performance for a collection larger than the GOV2. This should be addressed in future.

8 Conclusions and further work

We have presented and evaluated several skipping optimizations to pipelined query processing. For $\mathrm{SLT_N}$ and $\mathrm{SLT_S}$ our results indicate a significant improvement over the baseline approach. We also came up with a pruning approach ($\mathrm{MSD}^\star_\mathrm{S}$) that provides a result quality equivalent to a non-pruned evaluation, while having a considerably good performance. Further, we have observed that processing queries with conjunctive semantics (AND) provides good retrieval performance and efficient query processing. Although the state-of-the-art approach considers disjunctive (OR) queries, in future, we would like to take a closer look at AND queries. Finally, based on our current results, we outline three techniques that can further improve the performance of $\mathrm{MSD}^\star_\mathrm{S}$:

Dynamic load balancing. The load balancing of $\mathrm{MSD}^\star_\mathrm{S}$ can be improved by gradually moving the posting lists with the highest or the lowest $\hat{s}$ values to one of the neighbouring nodes. Compared to the previously presented fill-smallest and graph-partitioning techniques, this approach will reduce the network volume at repartitioning and can be done dynamically. In order to avoid moving data back-and-forth, we can further replicate the bordering posting lists and fine-tune partitions at the lexicon level, without actual repartitioning.

Hybrid query processing. $\mathrm{MSD}^\star_\mathrm{S}$ tends to place the shortest posting lists (corresponding to rare terms) on the first nodes. Therefore, by transferring these (complete) posting lists to a node corresponding to a later sub-query we can remove decompression, processing and accumulator transfer from the first few nodes (with an additional opportunity for parallelism). The node receiving the posting lists will in this case substitute an accumulator set with a few short

posting list. In order to minimize the load on the sending node and the overall network load, the nodes can further cache the received lists.

Intra-query concurrent processing. Document-at-a-time processing of sub-queries allows to transfer accumulators to the next node as soon as possible. This feature can be utilized to provide intra-query concurrency and improve the performance at low multiprogramming levels. In [6], we have already evaluated an extension to an earlier version of $\text{MSD}^{\star}_{\text{S}}$. The optimization splits the document ID range into several sub-ranges, called fragments, and does intra-query parallelization at the fragment level, both across the nodes and on the same node. The experiments [6] with a smaller subset of the TREC 2005 Efficiency Topics indicated that this optimization allows to reach a similar peak-throughput at nearly half of the latency. We assume this to be applicable to our current method, however a further evaluation for the 2006 topics is needed.

Acknowledgments. This work was supported by the iAd Centre and funded by the Norwegian University of Science and Technology and the Research Council of Norway.

References

1. S. Büttcher, C. L. A. Clarke, and G. V. Cormack. *Information Retrieval: Implementing and Evaluating Search Engines.* 2010.
2. B. B. Cambazoglu and C. Aykanat. A term-based inverted index organization for communication-efficient parallel query processing. In *IFIP NPC*, 2006.
3. S. Jonassen and S. E. Bratsberg. Impact of the Query Model and System Settings on Performance of Distributed Inverted Indexes. In *NIK*, 2009.
4. S. Jonassen and S. E. Bratsberg. A combined semi-pipelined query processing architecture for distributed full-text retrieval. In *WISE*, 2010.
5. S. Jonassen and S. E. Bratsberg. Efficient compressed inverted index skipping for disjunctive text-queries. In *ECIR*, 2011.
6. S. Jonassen and S. E. Bratsberg. Intra-query concurrent pipelined processing for distributed full-text retrieval. In *ECIR*, 2012.
7. N. Lester, A. Moffat, W. Webber, and J. Zobel. Space-limited ranked query evaluation using adaptive pruning. In *WISE*, 2005.
8. C. Lucchese, S. Orlando, R. Perego, and F. Silvestri. Mining query logs to optimize index partitioning in parallel web search engines. In *InfoScale*, 2007.
9. A. Moffat, W. Webber, and J. Zobel. Load balancing for term-distributed parallel retrieval. In *SIGIR*, 2006.
10. A. Moffat, W. Webber, J. Zobel, and R. Baeza-Yates. A pipelined architecture for distributed text query evaluation. *Inf. Retr.*, 2007.
11. T. Strohman, H. Turtle, and W. B. Croft. Optimization strategies for complex queries. In *SIGIR*, 2005.
12. H. Turtle and J. Flood. Query evaluation: strategies and optimizations. *Inf. Proc. and Manag.*, 1995.
13. W. Webber. Design and evaluation of a pipelined distributed information retrieval architecture. Master's thesis, 2007.
14. H. Yan, S. Ding, and T. Suel. Inverted index compression and query processing with optimized document ordering. In *WWW*, 2009.
15. J. Zhang and T. Suel. Optimized inverted list assignment in distributed search engine architectures. In *IPDPS*, 2007.