Interval-Based Pruning for Top-k Processing over Compressed Lists

Kaushik Chakrabarti #1, Surajit Chaudhuri #2, Venkatesh Ganti *3

#*Microsoft Research, USA*
{ 1kaushik, 2surajitc} @microsoft.com

**Google Inc., USA*
3vganti@google.com

***Abstract*—Optimizing execution of top-k queries over record-id ordered, compressed lists is challenging. The threshold family of algorithms cannot be effectively used in such cases. Yet, improving execution of such queries is of great value. For example, top-k keyword search in information retrieval (IR) engines represents an important scenario where such optimization can be directly beneficial. In this paper, we develop novel algorithms to improve execution of such queries over state of the art techniques. Our main insights are pruning based on fine-granularity bounds and traversing the lists based on judiciously chosen "intervals" rather than individual records. We formally study the optimality characteristics of the proposed algorithms. Our algorithms require minimal changes and can be easily integrated into IR engines. Our experiments on real-life datasets show that our algorithm outperform the state of the art techniques by a factor of 3-6 in terms of query execution times.**

I. Introduction

In many scenarios, top-k queries are processed over record-id ordered, block-compressed lists. One of the primary examples of this is top-k keyword search in information retrieval (IR) engines. Here, the user specifies one or more query terms and optionally a Boolean expression over the query terms and asks for k documents with the highest scores that satisfy the Boolean expression. The score of a document is typically computed by combining per-term relevance scores using a monotonic combination function.

To support such keyword search queries efficiently, IR engines maintain an inverted index which stores a "posting list" for each term. The posting list contains one *posting* $\langle docid, tscore \rangle$ for each document containing the term where $docid$ denotes the identifier of the document and $tscore$ denotes the term's contribution to the overall document score (referred to as term score for the document). The postings in each posting list are stored *in docid order* [17], [9], [18], [19], [4], [15]. In order to save disk space and reduce I/O cost, the posting lists are stored as a sequence of blocks where each block is compressed [20]. The compression techniques ensure that each block can be decompressed independently [17], [5], [19], [18]. Storing the postings in docid order allows: (i) better compression as d-gaps (difference between consecutive docids) can be encoded more compactly [18] (ii) more efficient list intersections [9] and (iii) efficient maintenance as new documents are added to the collection [20]. Due to the above reasons, majority of commercial IR engines like Lucene, Google Search Appliance, Microsoft Sharepoint Enterprise Search, Oracle Text Search and SQL Server Full Text Search store the posting lists in docid-ordered, block-compressed form.

Most modern IR engines use the *document at a time (DAAT)* framework to process queries [9], [16], [4], [18]. The naive way to answer top-k queries using DAAT is to traverse the posting lists of the query terms in tandem from one docid to the next. For each docid, DAAT will evaluate the Boolean expression, and if it is satisfied, will compute its final score. It will maintain the running set of k highest scoring documents seen so far and will return it at the end. The above algorithm, henceforth referred to as *naive DAAT* algorithm, scans and decompresses the *entire* posting list for each query term and computes the scores of *all* docids in the lists that satisfy the Boolean expression. Hence, it is inefficient for large document collections [16], [15]

A. Review of State-of-the-art

The state-of-the-art technique is to leverage the top-k constraint to "skip" over parts of the posting lists [16], [4], [15], [8].[1] This has proven useful in reducing the decompression and score computation costs incurred by the naive DAAT algorithm [15]. In this paper, we argue that the opportunities to skip can be expanded far beyond the state-of-the-art; this would result in significant performance gains.

We illustrate the state-of-the-art skipping algorithm through an example. Consider the freetext query (i.e., no Boolean expressions) containing query terms q_1 and q_2. The posting lists of the terms are shown in Figure 1. The horizontal axis represent the domain of docids. Each posting is shown as a dot and labeled with $\langle docid, tscore \rangle$. Each block is shown as a rectangle spanning from the minimum docid to the maximum docid in the block (here, each block contains 2 postings). For this example, we assume that the final document score is the sum of the individual term scores; it can be any monotonic function as discussed in Section III. The state-of-the-art algorithm maintains an upper bound score for each term: its maximal contribution to any document score. The term upper bounds for q_1 and q_2 are 5 and 6 respectively. Suppose $k = 1$. It opens a cursor on the posting list of each query term. Initially, both cursors point to docid 1. The

[1] The skipping algorithms used in commercial enterprise search engines are not known publicly. We discuss the state-of-the-art from the research literature.

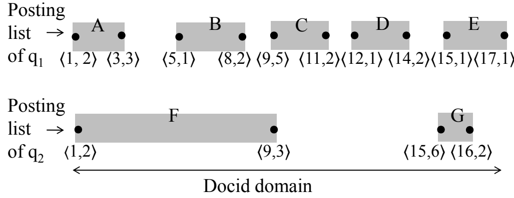


Fig. 1. Skipping using term upper bounds.

algorithm computes the score (4) for the docid 1. If the upper bound of a query term is less than the kth highest score seen so far, it skips parts of that term's posting list. This does not happen until the algorithm encounters docid 9. After it encounters docid 9, the kth highest score becomes 8. The cursors are pointing to docids 11 and 15 respectively. However, since the upper bound of q_1 (5) is less than 8, it concludes that documents containing only q_1 cannot be in the top-k result. So, it advances the cursor for q_1 to docid 15. Thus, it skips the block D, saving the costs of decompressing it and computing the scores of the documents in D. We refer to the above algorithm as *term upper bound-based skipping* (*TUB-skipping* in short).

As observed in [15], TUB-skipping provides limited pruning for a number of reasons. First, computing upper bounds at such a coarse granularity produces *weak bounds*. Even if there is a single docid in the posting list with a high score, the term upper bound score will be high. Weak upper bounds lead to limited skipping. Second, skipping is most beneficial when we skip entire blocks, not just individual documents. This is because block skipping saves both decompression and score computation costs while document skipping saves only the latter. Even if TUB-skipping finds a posting list to skip, it is not able to skip blocks unless the entire block of that posting list falls between two successive docids of the other posting lists (e.g., block D in the above example). This is unlikely unless one of the query terms are significantly more frequent than the other terms. The threshold family of algorithms are also not effective in this scenario; we discuss the details in Section II.

B. Our Contributions

To overcome the above shortcomings of TUB-skipping, we propose to *maintain upper bounds at a much finer granularity, specifically, at the granularity of blocks*. Since these bounds are much tighter, simply plugging in these bounds in place of the term upper bounds in the TUB-skipping algorithm will lead to better pruning. For example, after computing the score (4) of docid 1, the cursors point to docids 3 and 9 respectively. Since the upper bound score of block B (i.e., maximum term score of any document in the block) is 2 (≤ 4), we can advance the cursor to 9, thereby skipping block B. This was not possible in the original TUB-skipping algorithm since the term upper bound is 5 (> 4).

However, TUB-skipping *cannot take full advantage* of block-granularity upper bounds for skipping blocks. Continuing with the previous example, after advancing both cursors to docid 15, TUB-skipping would compute its score and hence end up decompressing blocks E and G. However, using the upper bounds of the blocks, we know that docids in the interval [15,16] can have maximum final score of $upperbound(E) + upperbound(G) = 1 + 6 = 7$. Hence, it is possible to skip block G (and also E by the same argument).

Designing algorithms to take advantage of the block-granularity upper bounds is challenging. Note that we cannot simply skip blocks based on the upper bound score of the block. We need to compute, from the block upper bounds, tight bounds on the *final* score (for the given query) of documents in a block and identify opportunities for skipping based on those bounds. Tightness of those bounds are crucial for effectiveness of skipping. This becomes complex because the blocks of different query terms are not aligned, i.e., they overlap partially with each other. Hence, different parts of the block can have different upper bound final scores. For example, block F has upper bound 6 for the part it overlaps with A, 3 where it overlaps with the gap and so on. We address this challenge by introducing the concept of *"intervals" of docids*. Our key idea is to use the block upper bounds to efficiently *partition the docid domain into a set of intervals* for the given query and compute a *single upper bound score* for each interval. Subsequently, we *traverse the posting lists at the granularity of intervals* and compute the top-k docids; we use the interval upper bounds to skip most of the intervals. This in turn leads to skipping of blocks. There are still several challenges: how to partition such that the upper bounds are tight and the query processing cost is minimized? how to compute such a partitioning efficiently? what is the best way to traverse the intervals? We address these challenges in this paper and identify a set of candidate algorithms. Our algorithms are simple and hence easy to integrate in IR engines using the DAAT query processing framework. We study the optimality characteristics of our algorithms. We identify an algorithm of choice that exploits available memory to process the intervals in a lazy fashion.

Several improvements have been suggested for TUB-skipping by leveraging "fancy lists" (a separately materialized list containing the few highest scoring documents for each term) [8], [15]. We show how our algorithms can also leverage fancy lists to improve performance. Furthermore, queries often specify Boolean expressions over the query terms. We present an optimization that can leverage such expressions to further improve performance.

Finally, we conduct a thorough empirical evaluation of our techniques over two real datasets, TREC Terabyte and TotalJobs CV corpus. Our experiments show that our techniques (i) decompress *one to two orders of magnitude* fewer blocks and (ii) compute scores of *one to two orders of magnitude* fewer docids compared with TUB-skipping techniques. This results in *3-6X improvement* in execution time compared with the state-of-the-art TUB-skipping technique.

To the best of our knowledge, this is the first paper to propose fine-granularity upper bounds and interval-based pruning. Like TUB-skipping algorithms, our algorithms are agnostic to the specific compression scheme used to compress

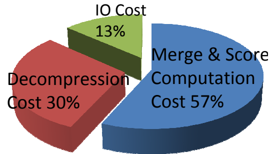


Fig. 2. Breakdown of the query processing cost of naive DAAT algorithm (averaged over 50 TREC Terabyte track queries on 8 million documents from TREC GOV2 collection).

the blocks. Like prior work, we focus on reducing the costs of decompressing the posting list (referred to as *decompression cost*) and that of performing docid comparisons and computing the final scores of documents (referred to as *merge and score computation cost*). These two represent the major components of the query processing cost ($> 85\%$) as shown in Figure 2. Although we present our techniques in the context of top-k keyword search in IR engines, our framework applies to other applications involving compressed lists like compressed data warehouses [14].

Outline: We discuss related work in Section 2. We present the preliminaries in Section 3. We describe our core algorithms in Section 4. In Section 5, we present two extensions: to leverage fancy lists and Boolean expressions. We evaluate our techniques empirically in Section 6 and conclude in Section 7.

II. Related Work

Several improvements have been proposed for the TUB-skipping technique by leveraging fancy lists [8], [15]. The main idea is to leverage the fancy list to compute tighter upper bound scores for each term. This results in better pruning. In our experiments, we compare our techniques with TUB-skipping techniques both with and without fancy lists. In both cases, our techniques significantly outperform TUB-skipping due to finer-granularity upper bounds and interval-based traversal. There is also a plethora of work on returning approximate top-k answers. We focus on exact top-k techniques in this paper. Furthermore, if the query involves a Boolean expression over the terms, it can also be used to skip blocks [10]. For example, if the Boolean expression is AND, we can skip a block that does not overlap with any block of any other posting list.

The threshold family of algorithms (TA) have been proposed to leverage the top-k constraint effectively in middleware systems [6], [11], [7]. However, TA requires the document list for each query term to be sorted by score. Since most commercial IR engines maintain the posting lists in docid-ordered, compressed form, obtaining the lists in score order requires *decompressing the entire posting list* and a *full sort* of the list. This is prohibitively expensive.

Another option is to maintain the posting lists in the score order instead of docid order [2], [3]. Subsequently, we can use TA without paying the above cost. However, this would require a complete change of the underlying index organization used in commercial IR engines. Furthermore, as observed in [8], such an approach can introduce other inefficiencies because compression is less effective as the d-gaps are larger compared to the docid-ordered case. Performing list intersections (for Boolean keyword queries) and updating the posting lists become more complex in this organization [20]. The blocked organization proposed in [3] where postings within block are docid-ordered while blocks are score-ordered addresses some of these issues. However, compression is still less effective compared to docid ordered lists and list intersections remain complex [3]. In this paper, we focus on techniques that require minimal changes to the underlying index used in commercial IR engines (i.e., docid-ordered, compressed posting lists).

Our pruning techniques are similar in spirit to multi-step k-nearest neighbor search over multidimensional index structures [13]. While the upper bounding principle still applies, the techniques (i.e., how the filter distances are computed, how the index is traversed) cannot be directly applied to our setting as query processing over inverted indexes is fundamentally different from that over multidimensional indexes.

III. Preliminaries

A. Query Model

We assume that the query contains m terms $q_1, q_2, \ldots, q_m$. The query can optionally specify a Boolean expression over the query terms. The system returns the k highest scoring documents that satisfy the Boolean expression (ties broken arbitrarily). We assume the score of a document d is $f(tscore(d, q_1), \ldots, tscore(d, q_m))$ where $tscore(d, q_i)$ denotes the per-term score of d for term q_i and f is a given *monotone* function. A function f is said to be monotone if $f(u_1, \ldots, u_m) \geq f(v_1, \ldots, v_m)$ when $\forall i, u_i \geq v_i$. Many of the popular IR scoring functions like TF-IDF, BM25 and BM25F fall in this class of scoring functions.

B. Data Model

We assume that a posting list is maintained for each term. The postings in each list are ordered by docid and organized as a sequence of compressed blocks as shown in Figure 4 [17], [5], [19], [18]. Each block typically contains a fixed number of consecutive postings (e.g., around 100 as discussed in [17]). Within a posting list, we identify a block by its *block number*. Each block is compressed and hence can be decompressed *independently*. We assume two modes of access to the blocks: *sequential* and *random* access.

Modern IR engines often maintain a *fancy list* for each term, i.e., a small fraction ($\sim 1\%$) of docids with the highest scores for the term [8], [15], [9]. These docids are removed from the blocks and are stored at the start of the posting list as shown in Figure 4. We show how we can leverage fancy lists in Section V-A.

C. Cost Model and Problem Statement

In the DAAT framework, the query processing cost has 3 components as shown in Figure 2: (i) *I/O Cost*: that of scanning the posting list of each query term on disk and transferring it to memory (ii) *Decompression cost*: that of decompressing the posting lists and (iii) *Merge and Score computation (M&S) cost*: that of performing docid comparisons and computing the final scores of documents. Given a keyword query, our goal is to return the top k docids and their

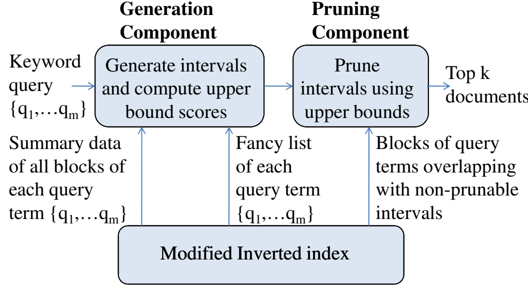


Fig. 3. Architecture of our approach

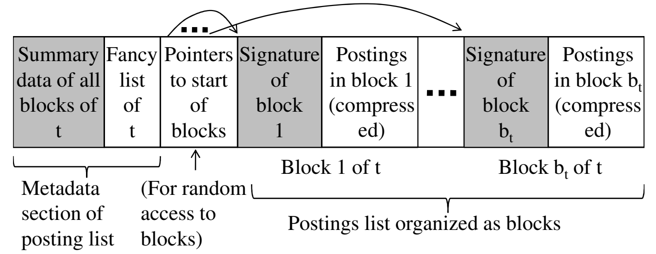


Fig. 4. Posting list of a term t. Shaded regions represent our modifications.

scores while incurring the minimum I/O, decompression and M&S costs.

IV. Our Approach

A. Overview of our approach

Figure 3 shows the architecture of our approach. It has 3 components:

- **Inverted Index Modification** (index build-time): We compute a *small amount of summary information for each block of each term* (including the upper bound) and store it in the inverted index.
- **Interval Generation** (query-time): Using *only* on the summary information, we partition the docid domain into a set of intervals and compute a single tight upper bound (on the final score of documents) for each interval.
- **Interval Pruning** (query-time): We then traverse the posting lists of the query terms at the granularity of intervals and compute the top-k docids. We use the upper bounds of the intervals to "prune" out most of the intervals. This leads to skipping of blocks.

We discuss the three components in detail the next three subsections.

B. Inverted Index Modification

We modify the posting list of each term in two ways as of shown in Figure 4:

Summary information for each block: While building the inverted index, we compute the following summary information for each block: the minimum docid, maximum docid and maximum term score of any document in the block. We refer to them as the *mindocid*, *maxdocid* and *ubtscore* (short form for upper bound term score) of the block respectively. We need to retrieve the summary data of *all* blocks of a term without accessing the blocks themselves, so we store the summary information for all blocks at the start of the posting list (as shown in Figure 4). IR engines already store some metadata at the start of the posting list (e.g., number of documents, term upper bounds, fancy list); we simply append our summary information to the existing metadata. Further, we store the summary data of the blocks in the same order the blocks are stored; this ensures efficient interval generation. The summary information is about 1.5% of the size of posting list (3 integers per block while block stores 200 integers assuming 100 postings per block). We compress the summary information using the same scheme used to compress the blocks.

Signature for each block : We compute a compact *signature* for each block that describes the postings in the block. We store it at the start of the block before the compressed postings as shown in Figure 4. If the query specifies a Boolean expression, we can use the signature for further pruning. We discuss the details in Section V-B.

Note that our approach has *almost negligible storage overhead*. Furthermore, the basic organization of the posting list remains unchanged, hence our approach retains all the desirable properties of the docid-ordered, block-compressed organization.

C. Interval Generation

C.1 Optimal Set of Intervals

Our goal is to partition the docid domain into a set of intervals such that we can compute a *single upper bound score* (of the *final* score of documents) for each interval. Tightness of these bounds are crucial for effective pruning; hence, we require that upper bound score to be the *tightest possible upper bound score* derivable from the summary information for *all* docids in the interval. Formally, the tightest possible upper bound score of a docid d (henceforth referred to as simply upper bound score of d) is $f(d.ubscore_1, \ldots, d.ubscore_m)$ where $d.ubscore_i$ is

$$\begin{aligned} &= b.ubtscore \text{ if there exists a block b in } q_i's \text{ posting list} \\ &\quad \text{such that } b.mindocid \leq d \leq b.maxdocid \\ &= 0 \text{ otherwise} \end{aligned}$$

In other words, we require the upper bound scores of all docids in an interval to be the same.

Definition 1: **(Intervals)** An interval v is a contiguous range of docids in the docid domain such that the upper bound scores of all docids $d \in v$ in the range are equal. ■

However, tightness is not the only factor. For example, one can generate one interval for each docid; this trivially satisfies the above definition. However, this would make interval pruning expensive since the latter needs to perform a check for each interval: whether it can be pruned or not. Hence, the optimal set of intervals is the partitioning that produces *the smallest number of intervals*.

Definition 2: **(Optimal Set of Intervals)** Among all possible sets of intervals that partition the docid domain, the optimal set is the one with the smallest number of intervals. ■

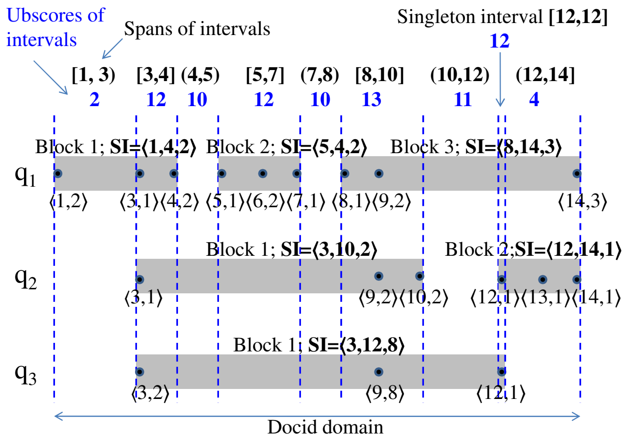


Fig. 5. Intervals for a freetext query with 3 terms q_1,q_2 and q_3. Summary information (SI) for each block, spans and ubscores of intervals are shown in bold.

Example 1: Consider a freetext query with three terms q_1, q_2 and q_3 whose posting lists are shown in Figure 5. Each block here contains three postings. The summary information ⟨ mindocid, maxdocid, ubtscore ⟩ is shown above each block. Figure 5 shows the 9 intervals for the query. The spans and ubscores of the intervals are shown on top. We denote a span by [], (), [) and (] depending on whether it is closed, open, left-closed-right-open or left-open-right-closed respectively. The span of the first interval is [1,3); all docids in this interval has upper bound score 2. ■

Observation: If we consider the docids in increasing docid order, the upper bound scores of docids change *only at the boundaries of the blocks* of the query terms. Hence, the boundaries of the blocks are also the boundaries of the optimal set of intervals.

C.2 Interval Generation Algorithm

We present an algorithm that exploits the above observation to efficiently generate the optimal set of intervals. Let $d_1, \cdots, d_l$ denote the mindocids and maxdocids of all the blocks of the query terms sorted in docid order. Since the mindocids and maxdocids of all blocks for each query term are stored in docid order in the inverted index, this can be obtained in time linear to the number of blocks in the query term posting lists (via a simple merge). Each pair $\{d_j, d_{j+1}\}$ of consecutive docids in the above sequence is an interval. For each boundary docid d_j, we need to identify which interval (the one corresponding to $\{d_{j-1}, d_j\}$ or the one corresponding to $\{d_j, d_{j+1}\}$) to include it in. There are 5 cases as shown in Figure 6:

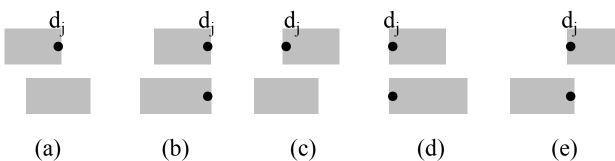


Fig. 6. Different cases for boundary points of intervals

- d_j is maxdocid of one or more blocks but not mindocid of any block (cases (a) and (b)): In this case, we include it in $\{d_{j-1}, d_j\}$ because d_j has the same upper bound score as other docids in that interval. Note that the upper bound score of the docids in the other interval is different from the upper bound score of d_j.
- d_j is mindocid of one or more blocks but not maxdocid of any block (cases (c) and (d)): We include it in $\{d_j, d_{j+1}\}$ by the same argument as above.
- d_j is mindocid of one or more blocks and maxdocid of one or more blocks (case (e)): In this case, we cannot include d_j in either the interval because the upper bound score of d_j is not the same as the upper bound scores of docids of either interval. So, we exclude it from both those intervals and generate a singleton interval $[d_j, d_j]$. $[12, 12]$ is an example of such a singleton interval in Figure 5.

We show how we can efficiently compute the upper bound score of an interval. Note that each interval overlaps with exactly one block or one gap for each query term. In the above algorithm, it is easy to keep track of the block/gap each interval overlaps with for each query term. The upper bound score $v.ubscore_i$ of an interval v for the ith query term is:

$$= \; b.ubtscore \text{ if } v \text{ overlaps with block } b \text{ for term } q_i$$
$$= \; 0 \text{ if } v \text{ overlaps with gap for query term } q_i$$

The ubscore $v.ubscore$ of the interval v is $f(v.ubscore_1, \ldots, v.ubscore_m)$.

We can formally show that the algorithm leveraging the observation stated in the previous subsection indeed produces the optimal set of intervals as defined in Definition 2.

Lemma 1: **(Optimality of Intervals)** *The above set of intervals is optimal.*

Performance overhead: The running time of the algorithm is linear to the number of blocks in the posting list of the query terms. Since the number of blocks is much fewer than the number of postings in the posting lists (around 1% assuming blocks with 100 postings), the interval generation component has negligible overhead (compared with the interval pruning component).

D. Interval Pruning

All our pruning algorithms follow the following template. They iterate over the intervals. For each interval, they perform the following check: can the interval be pruned? This involves checking whether the upper bound score (referred to as ubscore in short) of the interval is less than the kth highest docid score seen so far. If the check fails, they *"process"* the interval. Processing an interval involves reading the blocks overlapping with the interval from disk, decompressing them and running the DAAT algorithm over that interval.

The order in which we iterate over the intervals affect cost. For example, iterating over them in decreasing order of their ubscores is optimal in terms of number of intervals processed. This translates to superiority in terms of decompression and M&S costs. However, this involves random accesses to the blocks which is expensive. On the other hand, iterating over them in increasing order of their mindocids implies that the blocks are accessed via a sequential scan and hence has lower

Algorithm 1 PRUNESEQ

Input: List $\mathcal{V}$ of intervals, k
Output: Top k docids and their scores
Let `currblks` denote the decompressed blocks overlapping with current interval

```
1: for j = 1 to |V| do
2:   if V[j].ubscore > thresholdScore then
3:     for i = 1 to m do
4:       if V[j].blockNum_i ≠ GAP then
5:         if currblks[i].blockNum≠V[j].blockNum_i then
6:           currblks[i] ← Read V[j].blockNum_i) from disk (sequentially) and decompress
7:       else
8:         Clear block in currblks[i]
9:     ExecuteDAATOnInterval(currblks,V[j],currTopK,k)
10: return currTopK
```

I/O cost. We study the above two orders and then two other algorithms that tries to to balance between these two costs.

All our algorithms take as input the list of intervals $\mathcal{V}$ (in increasing order of their mindocids) and their ubscores. All our algorithms maintain the set `currTopK` of k highest scoring docids seen so far (initially empty). They also track the minimum score of current set of top-k docids, referred to as the *threshold score*.

D.1 Iterate over Intervals in Docid Order

We iterate over the intervals in increasing order of mindocids. The pseudocode of the algorithm, referred to as PRUNESEQ, is outlined in Algorithm 1. Note that $v.ubscore$ denotes the ubscore of the interval v and $v.blockNum_i$ denotes the block number (identifier) of the block overlapping with interval v for ith query term q_i. If v overlaps with a gap for q_i, we assign a special value (denoted by GAP) to $v.blockNum_i$. We discuss two steps in further detail:

Reading blocks (lines 3-8): Note that a block can overlap with multiple intervals. To avoid reading a block from disk and decompressing it multiple times, we read/decompress it once and retain it (in `currblks`) till we are sure that it is no longer needed, i.e., no subsequent interval overlaps with it.

Running DAAT on an interval (line 9): The ExecuteDAATOnInterval procedure takes the decompressed blocks of the interval and executes the DAAT algorithm over the span of the interval (i.e., from the mindocid of the interval till its maxdocid). We use the DAAT algorithm with the TUB-skipping optimization. If it finds any docid in that span with score higher than the threshold score, it updates `currTopK` accordingly. To locate the starting point in the blocks overlapping with the interval, we use binary search.

Example 2: We illustrate the execution of PRUNESEQ on the intervals shown in Figure 5. Suppose $k = 1$. PRUNESEQ will process all the intervals till [8,10]. At this stage, the threshold score is 12. So, it will prune the last 3 intervals. Thus, PRUNESEQ will avoid decompressing block 2 of q_2. ■

Note that PRUNESEQ accesses the blocks in the same order as they are stored in disk; so it performs a sequential scan over each query term posting list. Hence, it has the same I/O cost as TUB-skipping. However, it decompresses much fewer blocks and computes the score of much fewer docids compared with TUB-skipping. Hence, the overall cost is much lower (as confirmed by Figure 10(c)).

The main downside of PRUNESEQ is that initially, the threshold score may be a weak lower bound of the final top k documents. Hence, during this initial stage (until the threshold score becomes tighter), a large number of intervals may be processed unnecessarily (as evidenced by Figure 10(a)).

D.2 Iterate over Intervals in Ubscore Order

Alternately, we can iterate over the intervals *in decreasing order of their ubscores*. We process the intervals until we encounter an interval with ubscore less than or equal to the threshold score. At this point, we can terminate as none of the remaining intervals can contribute any docid to the top-k results. We refer to the above algorithm as PRUNESCOREORDER.

Note that since a block can overlap with multiple intervals, the above algorithm may end up accessing the same blocks on disk (random I/O) and decompressing them multiple times. We mitigate that cost by caching the decompressed blocks (referred to as `DCache`) during traversal.

PRUNESCOREORDER is optimal in terms of number of blocks decompressed over a fairly general class of algorithms. We formally capture this notion using instance-optimality [6]. Let $\mathcal{D}$ be the class of databases consisting of posting lists for each term. Let $\mathcal{A}$ be the class of algorithms that are allowed to perform either sequential or random accesses to blocks but use limited amount of buffers (that holds at most one block per query term). PRUNESCOREORDER is *instance optimal* over $\mathcal{A}$ and $\mathcal{D}$ with optimality ratio m (m is the number of query terms). This is, for every $A \in \mathcal{A}$ and $D \in \mathcal{D}$, $cost(\text{PRUNESCOREORDER}, D) \leq m \cdot cost(A, D) + c$ where c is a constant.

Theorem 1: **(Instance Optimality of PRUNESCOREORDER)** PRUNESCOREORDER is instance optimal over $\mathcal{A}$ and $\mathcal{D}$ in terms of distinct blocks decompressed with optimality ratio m.

Proof Sketch: It is easy to show that PRUNESCOREORDER only processes intervals with ubscore greater than final kth highest score. Using the notion of r-optimality [13], it can be shown that any algorithm $A \in \mathcal{A}$ cannot prune any such interval v, otherwise there will be false dismissals for some $D \in \mathcal{D}$. This implies A must decompress at least one block overlapping with each interval processed by PRUNESCOREORDER. Hence, the optimality ratio of m. ■

Although PRUNESCOREORDER is optimal in terms of number of blocks decompressed (which translates to lower decompression and M&S costs), it typically has much higher I/O cost compared with PRUNESEQ as it performs random I/O while accessing the blocks on disk. This can offset the savings in decompression and M&S costs. Our experiments confirm that this is indeed the case (Figure 11(a)) Note that

Algorithm 2 PRUNEHYBRID

Input: List $\mathcal{V}$ of intervals in docid order, ρ , k
Output: Top k docids and their scores

```
1: V_top ← Top ρ × |V| intervals ordered by ubscore
2: for j = 1 to |V_top| do
3:    if V_top[j].ubscore ≤ thresholdScore then
4:       return currTopK
5:    Clear blocks in currblks
6:    for i = 1 to m do
7:       if V_top[j].blockNum_i ≠ GAP then
8:          currblks[i] ← Lookup block V_top[j].blockNum_i in
            DCache, else read from disk and decompress (and
            update DCache) if not found
9:    ExecuteDAATOnInterval(currblks,V[j],currTopK,k)
10: Process unprocessed intervals in docid order as in PRUNESEQ
11: return currTopK
```

if the posting lists of the query terms reside completely in memory, PRUNESCOREORDER is optimal in terms of overall cost.

D.3 Iterate in Ubscore Order, then in Docid Order

We present an algorithm that judiciously trades off I/O cost with decompression/M&S costs to reduce the overall cost. The main idea is to process *a small number of intervals in ubscore order*, then switching and iterating over the rest of them *in docid order*. Often, the docids from the intervals processed during the first phase builds up a "good" set of current top-k documents, making the threshold score a tight lower bound of the final top k documents. In such cases, we can prune out more intervals during the second phase compared to PRUNESEQ, thus saving decompression and M&S costs. If the extra cost of random accesses during the first phase is lower than the savings in the decompression and M&S costs, the algorithm will have lower overall cost. The pseudocode of the above algorithm, referred to as PRUNEHYBRID, is outlined in Algorithm 2.

ρ is the fraction of intervals to be processed in score order before the switch is made; it is an input parameter to PRUNEHYBRID. The system designer can control the tradeoff using ρ. Note that PRUNEHYBRID reduces to PRUNESEQ when $\rho = 0$ and to PRUNESCOREORDER when $\rho = 1$.

Example 3: We illustrate the execution of PRUNEHYBRID on the intervals shown in Figure 5. Suppose $k = 1$ and $\rho = 0.1$. PRUNEHYBRID will process the top $0.1 \times 9 \approx 1$ intervals, i.e., [8,10] in score order. At this stage, the threshold score is 12. PRUNEHYBRID will consider the remaining intervals sequentially and will prune all of them. Thus, PRUNEHYBRID will avoid decompressing blocks 1 and 2 of q_1 and block 2 of q_2. ■

D.4 Process Intervals Lazily Using Available Memory

Since the cost of random access is much higher than that of sequential access, the optimal algorithm is to *process the intervals in score order* but *read the blocks in docid order*. In that way, we perform just one sequential scan (i.e., optimal I/O cost) and, at the same time, incur optimal decompression and M&S costs.[2] Our main insight is to *utilize the available amount of memory* to approach the above goal.

Suppose we have enough memory available to hold all the compressed blocks of the query terms. We can iterate over the intervals *in docid order* as in PRUNESEQ. We prune the intervals with ubscore less than or equal to threshold score as before. For the non-prunable intervals, instead of decompressing the blocks and processing the interval right away, we read the compressed blocks from disk and fetch them to memory. We refer to this as the *"gathering" phase*. At the end, we process the gathered intervals *in score order*. We refer to this as the *"scoring" phase*. This algorithm is optimal as defined above.

In practice, we may not have enough memory available to hold all the compressed blocks of the query terms. In that case, we perform the above "gathering" in *batches of intervals*. For each batch, we then process the intervals in score order. The pseudocode of the algorithm, referred to as PRUNELAZY, is outlined in Algorithm 3. We keep track of the the last interval `lp` processed or pruned during the last scoring phase and the last interval `lg` gathered during the last gathering phase.

Gathering phase (lines 2-12): During the gathering phase, we gather compressed blocks of the non-prunable intervals (starting from the first unprocessed interval (`lp`+1)) into a memory buffer. Once the buffer becomes full, we record the last interval we have gathered in `lg` and switch to the scoring phase.

Scoring phase (lines 13-25): During the scoring phase, we process the gathered batch of intervals, i.e., from (`lp`+1) till `lg` in *score order*. However, unlike in PRUNESCOREORDER, when a block of the interval is not found in the `DCache`, we *do not read from disk*; we read it from the memory buffer, decompress it, use it and cache it. When the termination condition is satisfied, we discard all the gathered blocks, record (in `lp`) that all intervals till `lg` has been processed or pruned and switch back to the gathering phase. We keep toggling between the two phases till we iterate over all the intervals.

Example 4: We illustrate the execution of PRUNELAZY on the intervals shown in Figure 5. Suppose $k = 1$. Let M denote the size of the memory buffer in terms of number of compressed blocks; suppose $M = 5$. PRUNELAZY gathers blocks of all intervals till (10,12); it then switches to the scoring phase. It processes [8,10] and prunes the rest. At this stage, the threshold score is 12. It then switches again to the gathering phase and gathers the remaining blocks. The next switch is back to the scoring phase during which it prunes all the intervals. Thus, PRUNELAZY will process only the interval [8,10] and hence avoid decompressing blocks 1 and 2 of q_1 and block 2 of q_2. ■

We formally show that if the upper bound scores of the intervals are *realizable upper bounds*, i.e., there exists at least one document in the interval with that score, PRUNELAZY *scales linearly to the memory budget*.

[2]PRUNESEQ does both in docid order while PRUNESCOREORDER does both in score order.

Algorithm 3 PRUNELAZY

Input: List $\mathcal{V}$ of intervals, memory budget M, k, `cursor`
Output: Top k docids and their scores
Let `membuf` denote the memory buffer
Let `lp` denote the last interval processed or pruned
Let `lg` denote the last interval gathered

1: `membuf` $\leftarrow \phi$, `lp` $\leftarrow 0$, `lg` $\leftarrow 0$
2: **GATHERINGPHASE:**
3: **for** j = (`lp`+1) to $|\mathcal{V}|$ **do**
4:   **if** $\mathcal{V}[j].ubscore >$ thresholdScore **then**
5:     **for** i = 1 to m **do**
6:       **if** $\mathcal{V}[j].blockNum_i \neq$ GAP and $\mathcal{V}[j].blockNum_i \notin$ `membuf` **then**
7:         **if** size(`membuf`) $\leq$ M **then**
8:           Read block $\mathcal{V}[j].blockNum_i$ from disk (sequentially) and add to `membuf`
9:         **else**
10:           `lg` $\leftarrow$ j-1
11:           **goto SCORINGPHASE**
12: `lg` $\leftarrow |\mathcal{V}|$
13: **SCORINGPHASE:**
14: $\mathcal{B} \leftarrow \{\mathcal{V}[j], j = \texttt{lp}+1, \cdots, \texttt{lg}\}$ ordered by ubscore
15: **for** j = 1 to $|\mathcal{B}|$ **do**
16:   **if** $\mathcal{B}$[j].ubscore $\leq$ thresholdScore **then**
17:     `lp` $\leftarrow$ `lg`
18:     Empty `membuf`
19:     **goto GATHERINGPHASE**
20:   Clear blocks in `currblks`
21:   **for** i = 1 to m **do**
22:     **if** $\mathcal{B}[j].blockNum_i \neq$ GAP **then**
23:       $currblks$[i] $\leftarrow$ Lookup block $\mathcal{B}[j].blockNum_i$ in `DCache`, read from `membuf` and decompress (and update `DCache`) if not found
24:   ExecuteDAATOnInterval($currblks$,$\mathcal{V}$[j],`currTopK`,k)
25: **return** `currTopK`

Theorem 2: **(Sensitivity of PRUNELAZY to memory)** *If the upper bound scores of the intervals are realizable upper bound scores and assuming that the docids are assigned randomly, the expected number of blocks decompressed by* PRUNELAZY *is* $\frac{\Sigma_i b_i}{M}$ *times the optimal number where* $\Sigma_i b_i$ *is the total number of blocks in the posting lists of the query terms and* M *denotes the size of the memory buffer (in terms of number of compressed blocks).*

Proof Sketch: Assuming the docids are randomly assigned, the expected number of intervals processed during each process phase is k. So, the expected total number of intervals processed is $k \cdot \frac{\Sigma_i b_i}{M}$. The expected optimal number of intervals processed is k. Hence, the proof. ∎

For $m = 1$, the upper bound scores are indeed realizable bounds. This may not be true $m \geq 2$. However, our experiments show that even for $m \geq 2$, the cost of PRUNELAZY scales linear to the amount of memory (Figure 11(c)).

Lemma 2: **(Correctness of algortihms)** PRUNESEQ, PRUNESCOREORDER, PRUNEHYBRID *and* PRUNELAZY *return correct top-k docids.*

V. EXTENSIONS

We first discuss how our algorithms can leverage fancy lists to improve performance. We then present an optimization to leverage Boolean expressions, if specified by the query, to further improve performance.

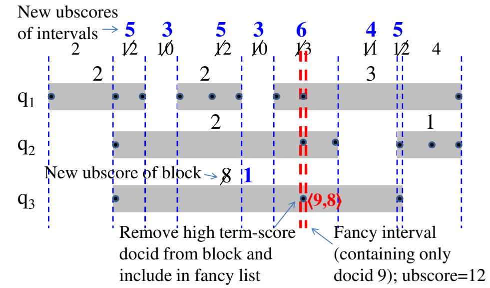


Fig. 7. Improving upper bound scores of intervals using fancy lists.

A. Adapting Interval Pruning to Leverage Fancy Lists

Many modern IR engines maintain a fancy list for each term: a small fraction ($\sim 1\%$) of documents with the highest scores for the term [8], [15], [9]. These docids are removed from the blocks and are stored at the start of the posting list as shown in Figure 4. Fancy lists are used to improve the term upper bounds used in TUB-skipping [8], [15]. If present, fancy lists can also improve the ubscores of the intervals and hence the performance of our interval-based algorithms. Consider the intervals for the freetext query with 3 terms q_1, q_2 and q_3 shown in Figure 5. If we remove the high term score docid 9 from q_3's posting list (and include in q_3's fancy list), the ubscores of the intervals decrease significantly as shown in Figure 7. We need to adapt our algorithms to take advantage of the fancy lists.

Adapting interval generation algorithm: In addition to the regular intervals, we generate a *"fancy interval"* f_d for each docid d in *any* of fancy lists of the query terms. Figure 7 shows the fancy interval for the docid (9) in q_3's fancy list. We obtain the ubscore of f_d as follows. If d is in the fancy list of q_i, we get the ubscore $f_d.ubscore_i$ of f_d for q_i from the fancy list. Otherwise, we get it from the block or gap d overlaps with for q_i. The overall ubscore $f_d.ubscore$ is $f(f_d.ubscore_1, \ldots, f_d.ubscore_m)$.

Adapting interval traversal and pruning algorithm: The traversal and pruning algorithms work as before over the combined set of regular intervals and fancy intervals. Only the DAAT processing of a fancy interval f_d is slightly different from a regular interval: for the term(s) for which d is a fancy hit, we get the term score from the fancy list itself.

B. Exploiting Boolean Expressions

Often, the query specifies a Boolean expression (e.g., AND) over the query terms. In such cases, the interval generation algorithm outputs only the intervals that contains documents that *can* satisfy the Boolean expression. For example, for AND query, we output only the intervals that overlap with a block for each query term, i.e., [3,4],[5,7],[8,10], [12,12] and (12,14) in Figure 5.

Often, an interval has blocks overlapping for all the query terms but contains no docid that satisfies the AND Boolean expression. [5,7] is such an interval in Figure 5. We present an optimization that avoids processing such intervals. We

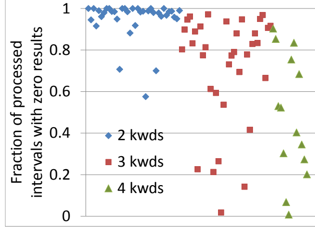


Fig. 8. Fraction of processed intervals with no results

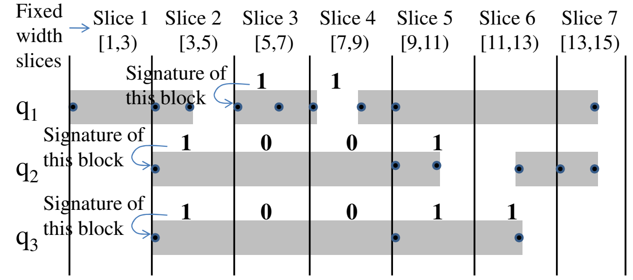


Fig. 9. Example signature scheme.

compute and store a *"signature"* for each block that contains information about the block at a finer granularity. We can then modify the interval traversal and pruning algorithm as follows: before processing a non-prunable interval, we use the signatures of the blocks overlapping with interval to check whether the interval may have docids that satisfy the Boolean expression. *We process the interval only if the check is satisfied*. If this check is significantly faster than processing the interval and the check fails for most intervals, it leads to lower cost.

However, we found that, for many queries, most intervals pass the check. Figure 8 shows the fraction of intervals containing at least one docid that satisfies the Boolean expression for a sample of 100 real-world queries on the 55GB document collection. This fraction is often high: it is above 90% for most queries with 2 terms and for several queries with 3 terms. All these intervals will obviously pass the check. Furthermore, for most signature schemes, the cost of performing the check, although lower than processing an interval, is not negligible. In such cases, performing the check on *all* non-prunable intervals is an overkill and can increase the overall cost.

To address this problem, we divide the non-prunable intervals into two categories: ones that are likely to fail the check and ones that are not. For each interval in the first category, we perform the check and process it only if the check is satisfied. For each interval in the second category, we skip the check and process it directly. Note that no matter how we divide the non-prunable intervals into these two categories, *correctness is not affected*: we always return exact top-k results. We first present an example signature scheme and then present *a novel, cost-driven technique to select the non-prunable intervals for which the check is performed*. For the purpose of exposition, we describe our techniques in the context of AND Boolean expression.

Example signature scheme: We present a simple signature scheme; any scheme can be used as long as the check has *no false negatives*. We first partition the domain of docids into fixed-width *"slices"*. Figure 9 shows the slices with width 2. Each block overlaps with a set of these (consecutive) slices. For example, block 2 of q_1 overlaps with slices 3 and 4. We compute a bitvector for each block with one bit corresponding to each slice it overlaps with. The bit is set to true if the block contains a docid in that slice and false otherwise. The bitvector is the signature of the block. The signatures of 3 blocks are shown in Figure 9. To perform the check on interval v, we take, for each block overlapping with v, the "portion" of the bitvector overlapping with v and perform a bitwise-and. If at least one bit in the result is set, the check is satisfied. To perform the check on interval [5,7], we take the portions from the above three blocks (11, 00 and 00 respectively). The check (bitwise-and of 11,00 and 00) would return false.

Selecting the intervals for which signature check is performed: For a non-prunable interval, we quickly estimate the probability of the check being satisfied. We perform the check *only if* that estimated probability is below a threshold θ; otherwise, we skip the check and process the interval directly. The two challenges are (i) estimating the probability of the check being satisfied *efficiently* and (ii) determine the best value for θ. Note that the choice of θ affects the query processing cost but not the correctness of the top-k results.

We estimate the probability using only the *fraction of 1s* in the signature of a block as follows. Let $d(b)$ denote the fraction of bits in the signature of block b that are set to 1 and $w(v)$ denote the width of interval v. Assuming the bits in a signature of a block is equally likely to be set, the probability of a bit in the signature being set is $d(b)$. Assuming independence among query terms, the probability of a bit in the result of bitwise-and being set is $\prod_i d(v.blockNum_i)$. The number of bits for the interval is $\frac{w(v)}{r}$. So, the probability of the check being satisfied (i.e., at least one of the bits is set) is $1 - \Big(1 - \prod_i d(v.blockNum_i)\Big)^{\frac{w(v)}{r}}$.

We present a cost-driven approach of determining the best value of θ: we model the query processing cost and select the θ that minimizes the expected query processing cost.

Theorem 3: The optimal threshold value θ_{opt} is $(1 - \lambda)$ where λ is the ratio of the cost of the check to cost of processing the interval.

Proof: Let $e(v)$ denote the estimated probability that interval v satisfies the check. Let $C_{ch}(v)$ and $C_{pr}(v)$ denote the cost of the check and that decompressing blocks and DAAT processing for an interval v; $C_{ch}(v) = \lambda \cdot C_{pr}(v)$. So, the cost is

$$P(e(v) \leq \theta) \cdot \Big(C_{ch}(v) + e(v) \cdot C_{pr}(v)\Big) + P(e(v) > \theta) \cdot C_{pr}(v)$$

$$= \Big(\lambda \cdot P(e(v) \leq \theta) + e(v) \cdot P(e(v) \leq \theta) + P(e(v) > \theta)\Big) \cdot C_{pr}(v)$$

Let $f(x)$ be the probability distribution of $e(v)$. The expected cost $E(\theta)$ is

$$\Big(\lambda \cdot \int_0^\theta f(x) + \int_0^\theta x f(x) + 1 - \int_0^\theta f(x)\Big) \cdot E(C_{pr}(v))$$

The expected cost is minimized when $\frac{dE(\theta)}{d\theta} = 0$, i.e.,

$$\lambda \cdot f(\theta_{opt}) + \theta_{opt} f(\theta_{opt}) - f(\theta_{opt}) = 0$$

Hence, the optimal threshold value θ_{opt} is $(1 - \lambda)$. ■

Thus, we perform the check on a non-prunable interval only if

$$1 - \left(1 - \prod_i d(v.blockNum_i)\right)^{\frac{w(v)}{r}} < (1 - \lambda)$$

Although we described our techniques in the context of AND Boolean expression, they can be easily generalized arbitrary Boolean expressions (arbitrary combinations of ANDs and ORs). We skip the details due to space limitations.

VI. Experimental Evaluation

We now present an experimental evaluation of the techniques proposed in the paper. The goals of the study are:

- To compare the performance of interval-based algorithms with TUB-skipping
- To compare the performance of the four traversal and pruning algorithms, PruneSeq, PruneScoreOrder, PruneHybrid and PruneLazy
- To compare the performance of interval-based algorithms with TUB-skipping in presence of fancy lists
- To evaluate the impact of the optimization for exploiting Boolean expressions.

A. Experimental Setting

Inverted Index Implementation

Since our algorithms require modifications to posting lists, we extract the posting lists out from an IR engine[3] , modify them as shown in Figure 4 and store them in binary files. We store the dictionary (containing each term in the collection, its IDFScore and a pointer to the start of its posting list) in an in-memory hash table with the term as key. Our algorithms rely on three inverted index APIs: `ReadMetaData` for reading the summary data of a given term (used by interval generation step) and `ReadBlockSeq` and `ReadBlockRand` for sequential and random access to a given block (used by traversal and pruning step). Our implementation of those APIs follow the traditional implementation (using standard file seek and binary read operations). We store 100 postings per block as proposed in [19]. We used variable-byte compression to compress the d-gaps and term scores inside the blocks. Variable-byte compression is commonly used in IR engines to compress posting lists (including in Lucene [1]) due to its fast decoding speeds [12]. Since the decompression cost is higher for other compression schemes [12], [20], we believe that our pruning techniques would provide even more savings for other schemes.

To study the performance with and without fancy lists and signatures, we created 3 versions of the index for each document collection: one without fancy lists and signatures, one with fancy lists but no signatures and one with both. For fancy lists, we use the top 1% of the docids in the posting list. For signatures, we use the bitvector scheme described in Section V-B. We use slice width of 100. We compressed the bitvectors using run-length encoding.

[3] From SQL Server Full Text Search (FTS) engine using the dynamic management function called `dm_fts_index_keywords_by_document`

TABLE I
Example queries from TREC Terabyte Track

volcanic activity	blue grass music festival history
domestic adoption laws	school mercury poisoning
ivory billed woodpecker	women rights saudi arabia
reverse mortgages	reintroduction gray wolves

Query Processing Algorithms

We implemented (i) naive DAAT (ii) TUB-skipping [4] and (iii) the four interval-based algorithms proposed in this paper. For PruneLazy, we use $M = 5000$ (size of memory buffer in terms of compressed blocks) unless otherwise mentioned. We also evaluate the above algorithms in presence of fancy lists and signatures. For all our experiments, we used a `DCache` of 1000 blocks (which is about 1MB). The above cache was shared across all queries.

Datasets

We report results over two real-life datasets.

TREC Terabyte: This is a collection of 8 million Web pages crawled from the .gov domain (part of the GOV2 collection). The total size of the collection is 201.3GB.

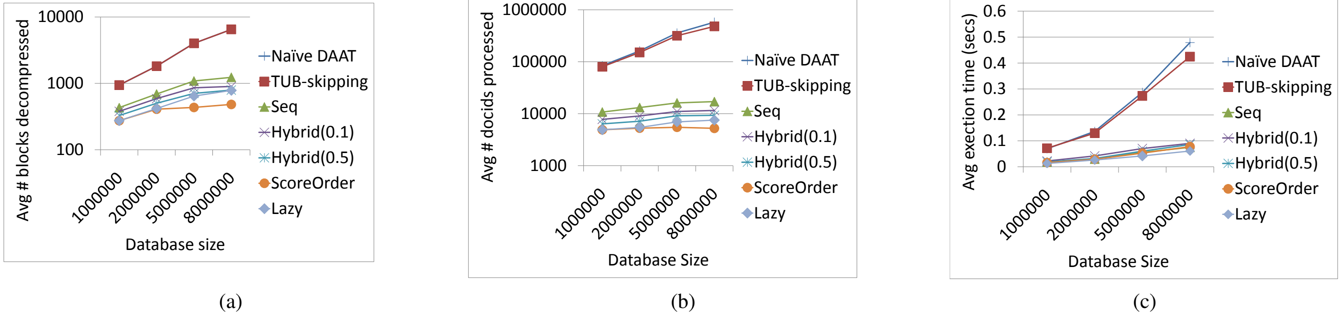
TotalJobs: This is a collection of 1118156 CVs (curriculum vitae) uploaded to TotalJobs.com. The total size of the collection is 54.4GB.

For the Trec dataset, our results are averaged over 50 topic (freetext) queries (751-800) from the 2005 TREC Terabyte Track (average query length is 2.875 words). A few example queries are shown in Table I. For TotalJobs dataset, we averaged across 500 real queries obtained from the TotalJobs.com query log (average query length is 2.74 words). We use the BM25 scoring function and $k = 10$ for all our experiments. All experiments were conducted on an AMD x64 machine with two 1.99GHz AMD Opteron processor and 8GB RAM, running Windows 2003 Server (Enterprise x64 edition). We conducted our experiments with a cold cache.

B. Summary of Results

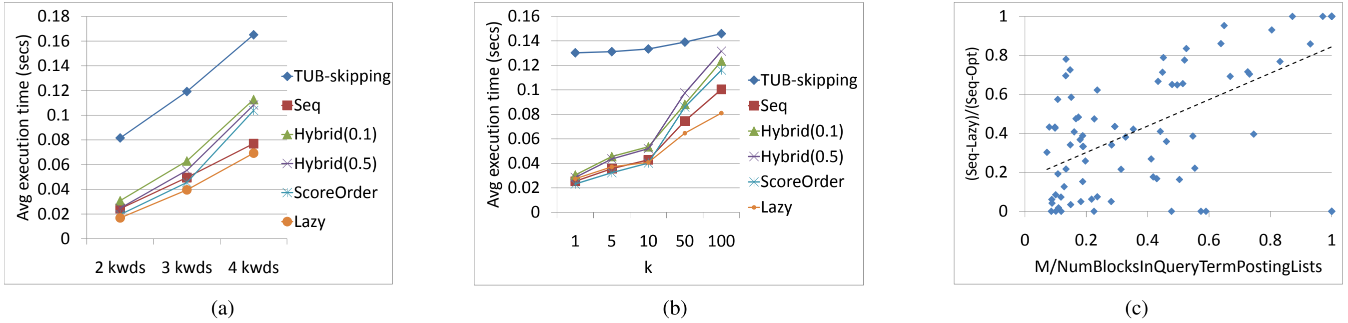
Performance on TREC Terabyte Dataset

We compare the various algorithms not only in terms of the overall execution time but also individual components of the cost, namely, the number of blocks decompressed and the number of documents processed. Figure 10(a) compares the four pruning algorithms with naive DAAT and TUB-skipping in terms of number of blocks decompressed. TUB-skipping hardly saves any block decompressions over naive DAAT; the plot for naive DAAT is not visible as it is superimposed by the plot for TUB-skipping. This validates our observation that TUB-skipping is unlikely to skip blocks unless one of the query terms is significantly more frequent than other terms; most queries in the TREC benchmark do not fall in this category (see Table I for examples). Our interval-based pruning algorithms decompress *one order of magnitude fewer*



(a) (b) (c)

Fig. 10. (a) Number of blocks decompressed w.r.t. database size (log scale) (b) Number of docids processed w.r.t. database size (log scale) (c) Execution time w.r.t. database size



(a) (b) (c)

Fig. 11. (a) Execution time w.r.t. number of keywords (b) Execution time w.r.t. k (c) Sensitivity of PRUNELAZY to memory

blocks compared with TUB-skipping. This is because the fine-granularity bounds and interval-based pruning that exploits those bounds leads to much better block skipping. Among the interval pruning algorithms, PRUNESCOREORDER is indeed the best (as formally stated in Theorem 1). We can see that PRUNELAZY is quite close to the optimal.

Figure 10(b) compares our pruning algorithms with naive DAAT and TUB-skipping in terms of the number of docids for which scores were computed. TUB-skipping is slightly better than naive DAAT; however, the skipping is limited due to weak upper bounds and docid-granularity traversal. Our interval-based algorithms compute scores for *one to two orders of magnitude fewer docids* compared with TUB-skipping. This confirms that the fine-granularity bounds and interval-based pruning leads to much better pruning. Among the interval pruning algorithms, PRUNESCOREORDER is the best as expected while PRUNELAZY is close to the optimal.

Figure 10(c) compares the various algorithms in terms of the overall execution time. The improvements in number of block decompressions and score computations translates to improvement in execution time: our algorithms *are 3-6 times faster* than TUB-skipping. PRUNELAZY performs the best among the interval pruning algorithms; it is *20-30% faster than other interval-based algorithms*. PRUNESCOREORDER is inefficient due to the high I/O cost. PRUNEHYBRID performs fewer random accesses compared with PRUNESCOREORDER but needs to also perform a sequential scan; the overall I/O cost is still high enough to offset the savings in decompression and M&S costs. We report the results for two values of the parameter ρ that controls the tradeoff (0.1 and 0.5); we obtained similar results for other values of ρ. PRUNELAZY outperforms PRUNESEQ as it has lower decompression and M&S costs compared with the latter and the same I/O cost.

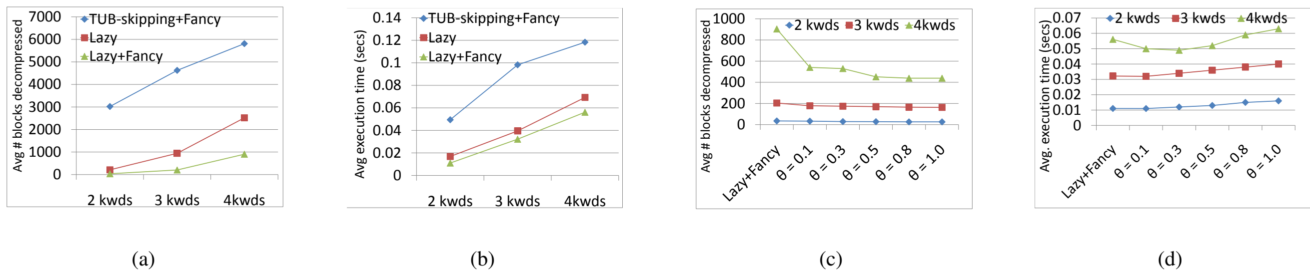
Performance on TotalJobs Dataset

Figure 11(a) compares our pruning algorithms with TUB-skipping in terms of the overall execution time. As in the TREC dataset, our interval-based algorithms are *3-6 times faster* than TUB-skipping. PRUNELAZY performs the best among the interval pruning algorithms; it is up to *a factor of 2 faster than other interval-based algorithms*.

Figure 11(b) compares the various algorithms in terms of execution time for various values of k. The cost of our pruning algorithms increase linearly with k. Higher the value of k, lower the threshold score, fewer the number of intervals pruned, hence higher the cost. Our interval-based algorithms outputperform TUB-skipping for all values of k.

Sensitivity of PRUNELAZY to memory

Figure 11(c) shows the sensitivity of performance of PRUNELAZY (in terms of the number of blocks decompressed) to the amount of available memory. The x-axis represents the fraction of blocks in the query term posting lists that fit in the available memory while the y-axis represents the improvement of PRUNELAZY over PRUNESEQ relative to the maximum possible improvement (i.e., of PRUNESCORE-ORDER over PRUNESEQ). The performance of PRUNELAZY *improves linearly with the amount of memory available*. We formally proved this behavior for single term queries (see Theorem 2); we empirically show that it holds for multi-term queries as well.



(a) (b) (c) (d)

Fig. 12. (a) # blocks decompressed when using fancy lists (b) Execution time when using fancy lists (c) Impact of signatures on # blocks decompressed (d) Impact of signatures on execution time

Impact of fancy list on performance

Figure 12(a) compares PRUNELAZY with and without fancy lists and TUB-skipping with fancy lists in terms of the number of blocks decompressed. Our pruning techniques significantly outperform TUB-skipping in presence of fancy lists as well. This implies that the benefits of fine-granularity bounds and interval-based pruning carry over to this case as well. Note that PRUNELAZY with fancy lists significantly outperform PRUNELAZY without fancy lists implying that that tighter block upper bounds leads to better pruning. Figure 12(b) compares the above techniques in terms of overall execution time. PRUNELAZY+FANCY is *3-5x faster* than TUB-skipping+Fancy. Using fancy lists improves the execution time by 30% over PRUNELAZY without fancy lists.

Impact of signature on performance

Figure 12(c) shows the impact of signatures in terms of number of blocks decompressed for various values of θ. Signatures do not have significant impact for 2 or 3 keyword queries because signature-check is not performed for most intervals. On the other hand, for 4 keyword queries, signatures reduce the number of blocks decompressed significantly. Higher the value of θ, higher the number of intervals for which the signature-check is performed, fewer the number of blocks decompressed. Figure 12(d) shows the impact of signatures in terms of execution time. For lower values of θ (< 0.3), the improvement in the number of blocks decompressed is worth the extra cost of performing the checks, hence the execution time drops for these values. This not the case for higher values of θ (> 0.3). Hence, the overall execution time follows a U-shaped curve with the optimum at around $\theta = 0.3$. This validates our checking algorithm presented in Section V-B: λ (the ratio of the cost of the check to the cost of processing the interval) is 0.6, so the optimal choice of θ is 0.4. which is close to the observed optimal value of θ.

VII. CONCLUSIONS

In this paper, we study the problem of top-k processing over record-id ordered, compressed lists. We develop algorithms that improve execution of such queries over the state-of-the-art techniques. Our main contributions are to use fine-granularity bounds and pruning based on judiciously-chosen intervals. Our empirical study indicate that our techniques outperform state-of-the-art techniques by a factor of 3-6. Our algorithms require minimal change to the underlying index organization and hence can be easily integrated into IR engines.

Our work can be extended in multiple directions. We considered a simple set of summary data per block; it might be possible to achieve much better pruning by exploiting other kinds of summary information. However, the challenge is to keep the overhead of the approach low. Extending the pruning techniques for complex scoring functions (e.g., machine-learned ranking) is also an open challenge.

REFERENCES

[1] Apache lucene - index file formats. *http://lucene.apache.org/java/2_3_2/fileformats.html*, 2008.
[2] V. Anh and A. Moffat. Pruned query evaluation using pre-computed impacts. In *SIGIR*, 2006.
[3] H. Bast, D. Majumdar, R. Schenkel, M. Theobald, and G. Weikum. Io-top-k: index-access optimized top-k query processing. In *VLDB*, 2006.
[4] A. Broder, D. Carmel, M. Herscovici, A. Soffer, and J. Zien. Efficient query evaluation using a two-level retrieval process. In *Proceedings of CIKM Conference*, 2003.
[5] J. Dean. Challenges in building large-scale information retrieval systems (wsdm 2009 keynote). *http://research.google.com/people/jeff/WSDM09-keynote.pdf*.
[6] R. Fagin, A. Lotem, and N. Naor. Optimal Aggregation Algorithms for Middleware. *J. Comp. Sys. Sci.*, 66(4), 2002.
[7] U. Güntzer, W.-T. Balke, and W. Kießling. Optimizing Multi-Feature Queries for Image Databases. In *VLDB Conf.*, 2000.
[8] X. Long and T. Suel. Optimized query execution in large search engines with global page ordering. In *VLDB*, 2003.
[9] C. D. Manning, P. Raghavan, and H. Schtze. *Introduction to Information Retrieval*. Cambridge University Press, New York, NY, USA, 2008.
[10] A. Moffat and J. Zobel. Self-indexing inverted files for fast text retrieval. *ACM Trans. Inf. Syst.*, 1996.
[11] S. Nepal and M. Ramakrishna. Query Processing Issues in Image (Multimedia) Databases. In *ICDE Conf.*, 1999.
[12] F. Scholer, H. Williams, J. Yiannis, and J. Zobel. Compression of inverted indexes for fast query evaluation. In *SIGIR*, 2002.
[13] T. Seidl and H. P. Kriegel. Optimal multi-step k-nearest neighbor search. In *SIGMOD*, 1998.
[14] D. Ślezak, J. Wróblewski, V. Eastwood, and P. Synak. Brighthouse: an analytic data warehouse for ad-hoc queries. *PVLDB*, 2008.
[15] T. Strohman, H. Turtle, and B. Croft. Optimization strategies for complex queries. In *Proceedings of ACM SIGIR conference*, 2005.
[16] H. Turtle and J. Flood. Query evaluation: strategies and optimizations. *IPM*, 31(6), 1995.
[17] I. H. Witten, A. Moffat, and T. C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images*. Morgan Kaufmann, 1999.
[18] H. Yan, S. Ding, and T. Suel. Inverted index compression and query processing with optimized document ordering. In *In WWW Conf.*, 2009.
[19] J. Zhang, X. Long, and T. Suel. Performance of compressed inverted list caching in search engines. In *In WWW Conf.*, 2008.
[20] J. Zobel and A. Moffat. Inverted files for text search engines. *ACM Comput. Surv.*, 38(2), 2006.