On the Cost of Extracting Proximity Features for Term-Dependency Models

Xiaolu Lu
RMIT University
Melbourne, Australia
xiaolu.lu@rmit.edu.au

Alistair Moffat
The University of Melbourne
Melbourne, Australia
ammoffat@unimelb.edu.au

J. Shane Culpepper
RMIT University
Melbourne, Australia
shane.culpepper@rmit.edu.au

ABSTRACT

Sophisticated ranking mechanisms make use of term dependency features in order to compute similarity scores for documents. These features often include exact phrase occurrences, and term proximity estimates. Both cases build on the intuition that if multiple query terms appear near each other, the document is more likely to be relevant to the query. In this paper we examine the processes used to compute these statistics. Two distinct input structures can be used – inverted files and direct files. Inverted files must store the position offsets of the terms, while "direct" files represent each document as a sequence of preprocessed term identifiers. Based on these two input modalities, a number of algorithms can be used to compute proximity statistics. Until now, these algorithms have been described in terms of a single set of query terms. But similarity computations such as the Full Dependency Model compute proximity statistics for a collection of related term sets. We present a new approach in which such collections are processed holistically in time that is much less than would be the case if each subquery were to be evaluated independently. The benefits of the new method are demonstrated by a comprehensive experimental study.

Categories and Subject Descriptors

H.3.4 [**Information Storage and Retrieval**]: Systems and Software—*Performance evaluation*

Keywords

Experimentation; Measurement; Dependency Ranking Models

1. INTRODUCTION

A range of sophisticated ranking mechanisms have been proposed that make use of term dependency features in order to compute similarity scores for documents [16, 17, 19, 21]. The features that get used include exact phrase occurrences, in which a sequence of words appear consecutively; term bigram co-occurrences; and more generalized term proximity estimates. Intuitively, these methods build on the belief that if multiple query terms appear next to, or nearby each other, it can be inferred that the document in question has an increased probability of being relevant to the query compared to a document in which the occurrences are separated by many intervening words.

CIKM'15 , October 19–23 2015, Melbourne, VIC, Australia

ACM 978-1-4503-3794-6/15/10 ...$15.00.
http://dx.doi.org/10.1145/2806416.2806467.

Here we examine the processes that are used to compute the necessary frequency statistics, with an emphasis on generalized term proximity features. For example, the "unordered window" attribute is based on, for each document being ranked, the number of distinct intervals of some specified length within the document in which all of the listed query terms appear. Two distinct input structures can be used to compute proximity features – inverted files, in which term positional information is retained as part of the posting associated with each terms' presence in the document; and the "direct" files that are constructed by some retrieval systems, in which each document is represented by a surrogate consisting of a sequence of preprocessed term identifiers.

A number of algorithms have been described for computing proximity statistics using these two input structures. These approaches include the plane-sweep and divide-and-conquer methods described by Sadakane and Imai [18], and other similar approaches described by Huston [12] and Clarke et al. [8]. All of the algorithms are presented in terms of a single set of query terms, with the objective being to compute a list of minimal intervals in the given document within which each query term appears at least once. But similarity computations such as the Full Dependency Model (FDM) [16] involve the computation of proximity statistics for a collection of related term sets, because every subset of terms from the underlying query is permitted to influence the final score. That is, multiple related queries of the form considered by Sadakane and Imai must be evaluated as part of scoring each document of the collection in a ranked retrieval environment. These considerations lead us to two research questions.

RQ 1 *Do inverted lists consistently offer advantages for computing proximity statistics when compared to direct files?*

RQ 2 *Is it possible to compute multiple related proximity statistics within a single evaluation?*

In this paper we present a new approach in which a suite of related general proximity queries can be computed *in a single pass*, generating significant execution time savings compared to the alternative of evaluating each subset of the terms independently. By exploiting a key insight in the way optimal subquery intervals must be formed, we show how to dramatically reduce the cost of computing all of the intervals for all query term combinations. We show that FDM-like scoring can be carried out holistically in time that is much less than would be the case with independent subquery evaluations, thereby answering the second question in the affirmative. The experimental study we have undertaken also provides evidence that while inverted lists are often superior to direct files, this is not always the case.

2. BACKGROUND

Phrases and proximity in IR Many of the recent ranking models in information retrieval include proximity features, such as term-pairs, phrases or unordered windows. Early models such as INQUERY [6] considered both phrases and proximity features, and a number of studies have been carried out to better understand their relative contributions. For example, Hawking and Thistlewaite [10] investigated the value of phrase-only queries and showed that documents that score higher using only phrase features may in fact be less relevant, and that phrase features may not be sufficient to reliably increase effectiveness. Tao and Zhai [20] compare several different term proximity models to determine which combination of k-term proximities can best improve query effectiveness. After comparing all possible proximity feature mappings in a BM25 ranking model, they conclude that term-pair proximity is sufficient to obtain a statistically significant improvement in effectiveness.

The common thread in using either phrases or proximity is that effectiveness can be improved some of the time, but predicting which method works best is difficult. Based on this insight, Metzler and Croft [16] proposed a Markov Random Field (MRF) model that incorporates both types of dependency relations into a single weighted model. Term dependencies in the SDM (Sequential Dependency Model) and FDM (Full Dependency Model) are represented as matches against ordered or unordered windows, using the Indri[1] operators `#ow`λ and `#uw`λ. The basic MRF model improves effectiveness over a bag-of-words model, especially on large collections and long queries [14]. Effectiveness can be further improved by using a Weighted Sequential Dependency Model where concept weightings are determined in a supervised manner instead of using a fixed configuration [2, 13]. Huston and Croft [13] found that k-term proximity improves effectiveness for some queries, but not all. The exact interaction of term-pair versus k-term concept weightings remains inconclusive and an interesting area of further study.

Computing proximity features Extracting all proximity features is a computationally expensive task. As a consequence, most efficiency studies consider only term-pair proximity, since term co-occurrences can easily be computed on the fly, or pre-indexed. Asadi and Lin [1] show that extracting BM25 and all two-term dependency features requires longer (20.6 microseconds per document) than only considering unigram BM25 features (14.6 microseconds per document). Term co-occurrences can also be precomputed without the size of the index increasing too much. Based on ranking models proposed by Büttcher et al. [4], Broschart and Schenkel [3] designed a combined index structure for both terms and term-pairs. Their index can be tuned to balance efficiency and effectiveness. Yan et al. [22] also index term co-occurrences, but use a different ranking model and consider only term-pairs within a fixed distance. They also incorporated early termination techniques to improve efficiency when scoring documents. Rather than keeping exact position information in the index, Elsayed et al. [9] only store approximated counts for various term-pair features. They segment documents into different buckets and use a membership query to determine if two query terms appear in adjacent buckets. Elsayed et al. also examine several different strategies to partition documents, and different encodings to reduce the total index size.

Relatively few methods have been described for computing k-word proximity features when k is larger than two. Asadi and Lin [1] suggest that the problem of matching `#ow`λ and `#uw`λ can be formulated as a set intersection and evaluated using the *small adaptive* algorithm, but then mainly focus on extracting term co-occurrence features. Two $O(N \log N)$ approaches for finding optimal intervals in an N-word document that include a set of k query terms were proposed by Sadakane and Imai [18]: a method that they denoted as *plane-sweep*; and a method that they categorized as *divide-and-conquer*. Experiments conducted on data indexed using both suffix arrays and inverted files showed that unless one query term has a much lower term frequency (TF), plane-sweep is the best approach. Sadakane and Imai suggest that performance could be improved by selectively choosing plane-sweep or divide-and-conquer based on the relative term frequency, but do not explore this option in detail.

[1] `http://www.lemurproject.org/indri.php`

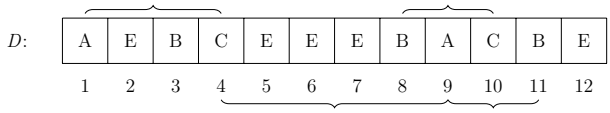

Figure 1: For document D and query $Q = \{\mathrm{A}, \mathrm{B}, \mathrm{C}\}$ there are four optimal 3-intervals: $1\ldots4$, $4\ldots9$, $8\ldots10$ and $9\ldots11$. The interval $3\ldots9$ is not optimal because it has $4\ldots9$ as a proper subset.

Contribution We focus on how to efficiently compute term proximity features for queries of $k \geq 3$ terms, assuming a context in which the $2^k - k - 1$ non-empty subsets of those terms must all be treated as *subqueries* for which term statistics are to be computed. This point of view allows us to develop efficient approaches to the holistic generation of *sets* of data for use in ranking computations, covering all of the requirements of an overarching k-term query. As is demonstrated in the development below, that point of view provides more efficient document scoring than is possible if standard interval-finding techniques are used in an iterative manner, processing each subquery separately.

We first revisit the problem of calculating k-word proximity by formalizing the problem. We next review current solutions according to their corresponding input structure – inverted indexes, or direct files. In the new work, we then discuss how to compute all k-intervals using a single-pass strategy in Section 4; and demonstrate the efficacy of those methods in Section 5.

Definitions First we define what is meant by an optimal k-word proximity interval, relative to a query $Q = \{q_1, q_2, \ldots, q_k\}$ with k query terms, and relative to a document $D[1\ldots N]$ containing a sequence of N words:

DEFINITION 1 (OPTIMAL k-INTERVAL). *An interval $p_l \ldots p_r$ bounded by left position p_l and right position p_r is* optimal *if all k terms in Q appear in $D[p_l \ldots p_r]$, and there is no proper subset of $D[p_l \ldots p_r]$ with the same property.*

Figure 1 shows an example document, and lists the four optimal 3-intervals induced by the query $\{\mathrm{A}, \mathrm{B}, \mathrm{C}\}$.

Markov Random Field term dependency implementations use the operator `#uw`λ to match windows containing all specified query terms within a distance λ [15]. In these models, all non-empty subqueries of Q are formed, and proximity statistics computed for each subquery. For example, the query $Q = \{\mathrm{A}, \mathrm{B}, \mathrm{C}, \mathrm{D}\}$ gives rise to one 4-item subquery; four 3-item subqueries; and six 2-item subqueries. The total number of subqueries induced by a k-term query is given by $2^k - k - 1$. The rapid growth as a function of k means that methods that compute intervals for all proximity statistics in a single pass are of interest, and gives rise to the challenge we consider:

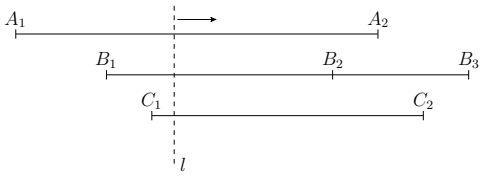

Figure 2: The plane-sweep algorithm checks whether the interval formed by the current leftmost representative and its next appearance encloses all the representatives. A virtual line *l* sweeps from left to right, with term positions determined using position offsets from an inverted file, or by sequentially processing a direct file.

PROBLEM 1 (OPTIMAL k-INTERVAL ENUMERATION). *For document $D[1 \ldots N]$ and query $Q = \{q_1, q_2 \ldots, q_k\}$, find all optimal $|Q'|$-intervals in D for all non-empty subqueries $Q' \subseteq Q$ in which $|Q'| \geq 2$.*

For example, for the query $Q = \{\mathrm{A}, \mathrm{B}, \mathrm{C}\}$ shown in Figure 1, the complete set of answer intervals required consists of optimal 2-intervals for $\{\mathrm{A}, \mathrm{B}\}$, $\{\mathrm{A}, \mathrm{C}\}$, and $\{\mathrm{B}, \mathrm{C}\}$, plus the optimal 3-intervals for $\{\mathrm{A}, \mathrm{B}, \mathrm{C}\}$ that have already been described.

Our goal in this paper is to compute the intervals for all of the subqueries in a single pass over the input structures, rather than require one pass per subquery.

3. SINGLE-QUERY INTERVAL FINDING

As a prelude to considering Problem 1, we describe methods for computing optimal k-intervals for one-off sets of terms.

Input representations Two alternative representations of positional information are used in IR systems. In a *direct file*, a surrogate for each document is maintained, with terms represented in document appearance order by their mapped identifiers. Each document in the collection has a contiguous section of the direct file allocated to it; and for each document that is to be scored in response to a query, that segment is identified and processed. The other option is to make use of a *positional inverted index*, which contains a postings list for each term in the collection. In this representation, the position offsets for each term are stored relative to either the start of the collection, or relative to the start of the enclosing document. Query processing is done via the postings lists, and only when a set of documents comprising the query's answer has been determined is it necessary to access documents or their parsed surrogates. Both representations can be used to support phrase and proximity operations. For example, Terrier[2] uses a direct file to support operators beyond bag-of-words computations, while Indri uses a merge operation over position offsets stored in an inverted file to support similar operations.

Despite considerable early work on the role of proximity in effectiveness computations [7, 10, 11], few studies have explored efficient computation techniques, and those that do mostly focus on the case where k is 2 or 3 [5, 8]. Sadakane and Imai [18] considered k in the unbounded case, and compared the efficiency of several different methods. One of the approaches they evaluated is the *plane-sweep* algorithm; if the input is assumed to be provided as postings lists, then plane-sweep outperformed the alternative *divide and conquer* approach.

Eager Plane Sweep (EPS) The plane-sweep approach processes terms in document order, checking whether the interval formed by the current leftmost term encloses all of the other endpoints. For example, consider the scenario shown in Figure 2. The interval $A_1 \ldots A_2$ contains B_1 and C_1. Therefore, $A_1 \ldots C_1$ is an optimal 3-interval. The leftmost term (in the example, A_1) is then discarded, and B_1 becomes the leftmost term. Now the interval $B_1 \ldots B_2$ does not contain both of the remaining terms, so B_1 is discarded, and C_1 becomes the leftmost term. As it turns out, this is the left-hand anchor point of another optimal 3-interval, through to A_2.

The plane-sweep approach can be visualized as sweeping a line from left to right across term positions in the document, hence the name. If the positions are represented using positional postings lists, then next-leftmost representatives are easily selected as are their successors, and all of the required information can be kept up to date with relatively little effort. We call this the *eager* version of plane-sweep; Algorithm 1 presents it in detail.

Algorithm 1 Eager Plane Sweep, Single Interval (EPS-IF-S)

Input: A query Q of k terms, and sorted position lists of all terms $\{P_1, P_2, \ldots, P_k\}$
Output: A set of intervals $\mathcal{I}$
  $\mathcal{I} \leftarrow \{\}$
  for $i \leftarrow 1$ **to** k **do**
    curr_occ$[i] \leftarrow$FIRST(P_i)
  end for
5: **while** $(\max_{1 \leq i \leq k}$ *curr_occ*$[i]) < \infty$ **do**
    lsym $\leftarrow \mathrm{argmin}_{1 \leq i \leq k}\{$*curr_occ*$[i]\}$
    rsym $\leftarrow \mathrm{argmax}_{1 \leq i \leq k}\{$*curr_occ*$[i]\}$
    lpos $\leftarrow$NEXT(P_{lsym}) // returns ∞ if P_{lsym} is exhausted
    if *lpos* $>$ *curr_occ*[*rsym*] **then**
10: $\mathcal{I} \leftarrow \mathcal{I} \cup \langle Q,$ *curr_occ*[*lsym*]$\ldots$*curr_occ*[*rsym*]$\rangle$
    end if
    curr_occ[*lsym*] $\leftarrow$*lpos*
  end while
  return $\mathcal{I}$

The inverted index is presumed to support the two usual access operations: FIRST and NEXT. Both return the position of a term in the current document being processed. The k-element array *curr_occ* is used to track the current positions of all terms in the query. In addition, *lpos* maintains the next position of *lsym*. If *lpos* is greater than *curr_occ*[*rsym*], the current interval is emitted. Finally in each iteration, *curr_occ*[*lsym*] is updated to the new most recent position, *lpos*. When any of the position lists is exhausted, the algorithm terminates. Figure 3 gives a concrete example of this process, matching the schematic shown in Figure 2. As already described, an interval is identified in the first iteration, but not in the second iteration.

Since each of the postings lists in the inverted index is already sorted by term positions, Algorithm 1 requires $O(\log k)$ time per iteration, using two heaps of k items to identify maximal and minimal representatives at steps 6 and 7 respectively. All other operations require $O(1)$ time each. Hence, over the k terms, there is a total of $T_{Q,D} = \sum_{i=1}^{k} |P_i| \leq N$ postings to be processed, and the total execution time is $O(T_{Q,D} \log k) = O(N \log N)$ [18].

Lazy Plane Sweep (LPS) In the EPS approach, the leftmost position is used as a pivot, and the next occurrence of that term is used to determine if the pivot is the leftmost end of an interval. However if the input format is a direct file, the NEXT operation cannot be carried out in $O(1)$ time per call, making the eager updating of *curr_occ* possibly costly.

Instead, in the *Lazy Plane Sweep* approach the direct file is sequentially scanned, and intervals are formed as their right-hand ex-

[2] `http://terrier.org/`

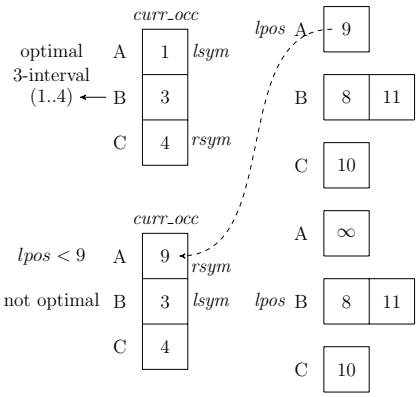

Figure 3: An example of Eager Plane Sweep. The array *curr_occ* records the current representative of each postings list. When $lpos > curr\_occ[rsym]$, as shown in the first configuration, an optimal 3-interval has been identified. The lower half then shows the following configuration, which does not generate an interval.

trema is processed. As with the EPS implementation, the most recent occurrence of each term is tracked in *curr_occ*. A sentinel value of $-\infty$ is used to indicate the absence of a term, and an interval can be formed only when a usable occurrence of each term is defined within the scope. That is, intervals are identified and emitted when the right-hand end of the interval is reached. If an interval is identified, the smallest of the values in *curr_occ* is then voided, as a signal that the next interval must contain a new instance of that term.

Algorithm 2 Lazy Plane Sweep, Single Interval (LPS-DF-S)

Input: A query Q of k terms, and a direct file $D[1 \ldots N]$
Output: A set of intervals $\mathcal{I}$

```
    I ←{}
    for i ← 1 to k do
        curr_occ[i] ←−∞
    end for
 5: for rpos ←1 to N do
        curr_occ[term_map(D[rpos])] ←rpos
        lsym ←argmin_{1≤i≤k}{curr_occ[i]}
        if curr_occ[lsym] > −∞ then
            I ←I ∪ ⟨Q, curr_occ[lsym]...rpos⟩
10:         curr_occ[lsym] ←−∞
        end if
    end for
    return I
```

Algorithm 2 provides details. Now it is the N values in the direct file corresponding to the document that are processed. At each iteration the next term (or non-term) $D[rpos]$ is considered (step 6). A mapping to query terms (denoted *term_map*() in the pseudo-code) is used to convert raw term numbers across the whole collection to restricted values pertinent to this query, with instances of the k query terms being converted (uniquely) to values in $1 \ldots k$, and instances of non-query terms being converted to (say) $k+1$. At each cycle, if inclusion of the symbol in $D[rpos]$ means that all of the k positions in *curr_occ* have positive values, then the right-hand end of an interval has been located. If that happens, the interval is saved, and then the left-hand symbol of it interval is voided (step 10), since it cannot take part in any further intervals.

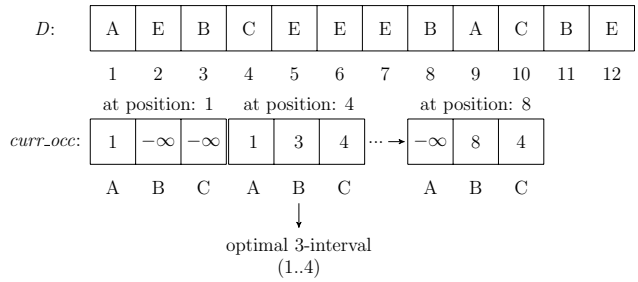

Figure 4: The LPS approach. Array *last_occ* of size k tracks the last position of each query term. When the minimum value in *last_occ* is $-\infty$, an optimal k-interval has not yet been formed. In the bottom array, an optimal k-interval has been formed, so the interval can be added into the result set based on the two endpoints.

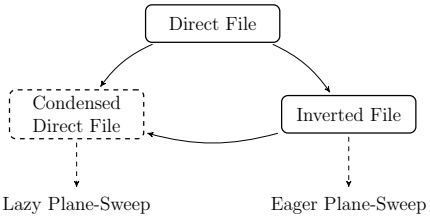

Figure 5: Input representations. A *condensed direct file* is a direct file restricted to the terms that appear in the query; it can be derived from a direct file in $O(N)$ time, or from an inverted file containing term positions in $O(T_{Q,D} \log k)$ time, where $T_{Q,D}$ is the total number of occurrences of query terms from Q in document D.

Step 7 in Algorithm 2 makes use of an "argmin" operator over k values, and is presented this way in order to draw out the symmetry between the EPS and LPS implementations. Step 7 does not, however, require a heap and $O(\log k)$ time per execution. That cost can be avoided by maintaining an *lpos* index into D, and advancing it to the right (at step 7) when necessary, bypassing any non-query terms. Each time *lsym* is required, it can then be computed as *term_map*($D[lpos]$). In total, over the whole of Algorithm 2, advancing *lpos* can require at most $O(N)$ time, and hence in an amortized sense, it is $O(1)$ time per loop iteration. The only other possible concern in Algorithm 2 is the cost of the term mapping *term_map*(); but it can be implemented as a hash-table lookup with expected time $O(1)$ per call. Hence, Algorithm 2 requires $O(N)$ time. Its correctness follows directly from the correctness of the eager plane-sweep approach [18].

Comparing Input Structures As has been noted, the two most common input representations are direct files and inverted indexes. Input via an inverted index is the "natural" arrangement for the eager plane-sweep process; conversely, the lazy plane-sweep approach was presented assuming input from a direct file. But the lazy mechanism can also be coupled with an inverted file – all that is required is to merge the postings lists for the query terms, and construct an intermediate form we call a *condensed direct file*, as shown in Figure 5. For query Q of k terms, and postings lists P_1 to P_k, the cost of doing the merge is given by $O(T_{Q,D} \log k)$, where $T_{Q,D} = \sum_{i=1}^{k} |P_k|$ is the sum of the lengths of the postings lists for D. So, the cost of generating the condensed direct file is the same as the cost of running EPS (Algorithm 1) starting with an in-

verted file. Note that the condensed direct file is query specific, and for that reason cannot be precomputed and stored.

Starting with a direct file, the specific condensed direct file for a query Q can be constructed in $O(N)$ time. Then, once the condensed direct file has been constructed, the EPS mechanism (Algorithm 2) can be executed in $O(T_{Q,D})$ time. Which of these various computation pathways should be chosen depends in no small part on the relative scale of $T_{Q,D}$, k, and N. For example, a short query of rare terms against a long document will favor the use of an inverted file/EPS approach; whereas a long query that includes one or more frequent terms is likely to execute more quickly using the direct file/LPS approach.

Either way, in the case of proximity queries in the context of FDM and similar retrieval models, multiple related subqueries must be evaluated. This need brings further tradeoffs, because it means that the cost of generating a condensed direct file can be amortized over the subqueries. This is one of the themes that is addressed in the next section.

4. EXTRACTING MULTIPLE INTERVALS

We now turn our attention to Problem 1, the larger task of computing interval statistics for a query Q and all of its subqueries Q', where $Q' \subseteq Q$ and $|Q'| \geq 2$.

Repeated Computation If a query has k terms, then there are $2^k - k - 1$ non-empty non-singleton term subsets possible: four, for a 3-term query; eleven, for a 4-term query, and so on. Hence, one obvious way of computing an all-subqueries solution is to iterate over the subsets, applying either Algorithm 1 or Algorithm 2 the requisite number of times. But this approach has the potential to be very expensive, and it would be better if the running time for generating a solution was dominated by polynomial functions of k and N, rather than being exponential in k. Moreover, in an ideal situation the size of the computed output set $\mathcal{I}$ should also be a factor that bounds the cost of the computation. That is, a useful algorithm is one that does the amount of work necessary to generate the required output, in this case $|\mathcal{I}|$, with an overhead cost as a function of N (or $T_{Q,D}$) and k that is additive rather than multiplicative.

One way in which time can be saved when multiple related subqueries Q' are being evaluated against the same document D has already been noted at the end of Section 3 – the formation of a query-specific condensed direct file (CDF) which retains only (and exactly) the positions of the terms in Q that appear in D. The cost of constructing the CDF is the same as running a single-query interval finding mechanism: $O(T_{Q,D} \log k)$ time if the input is an inverted file, or $O(N)$ time if the starting point is a direct file of length N, meaning that there is little point to the conversion operation for the single-interval task. On the other hand, the conversion cost can be amortized over multiple executions for the all-subqueries task, and allows a more flexible range of options. Hence, we can consider two relatively obvious baseline mechanisms for Problem 1, based on the two methods presented in Section 3:

- EPS-IF-S*: starting with an inverted file, apply Algorithm 1 a total of $2^k - k - 1$ times, reinitializing the postings list pointers and starting a fresh computation at each iteration.
- LPS-CDF-S*: starting with an inverted file or a direct file, construct a condensed direct file; then, using the CDF, apply Algorithm 2 a total of $2^k - k - 1$ times.

Both of these methods are used as reference points in the experimentation reported in Section 5 .

A Critical Observation Consider again the arrangement shown in Figure 2, and the occurrences of term B, marked by B_1, B_2, and B_3. Focusing on B_2, consider its relationship to the other two terms, A and C. Because there are no instances of term A that appear between B_1 and B_2, occurrence B_2 cannot be the right-hand end of an $\{\mathrm{A}, \mathrm{B}\}$ interval. On the other hand, the appearance of C_1 between B_1 and B_2 means that B_2 is the right-hand endpoint of a $\{\mathrm{B}, \mathrm{C}\}$ interval, spanning $C_1 \ldots B_2$.

Similarly, consider A_2, the next possible right-hand point for intervals encountered during the plane-sweep process. The previous occurrence of A is at A_1. There is no possible benefit from considering B_1 as a possible left-hand partner for A_1, because there is another instance of B between A_1 and A_2 that "shadows" it. Only the most recent instance of each symbol needs to be regarded as "interesting"; and B_2 is the instance of B that fits this definition, and defines the left-hand end of an $\{\mathrm{A}, \mathrm{B}\}$ interval. Position C_1 is also interesting, because it is the rightmost C prior to A_2, and hence can potentially be pinned at the left-hand end of intervals stretching through to A_2. Once that range $C_1 \ldots A_2$ is established as a viable one, all subsets of the set of distinct items strictly contained within that range need to be used to create intervals. In the example, B_2 lies between C_1 and A_2, and hence there are two subsets to be considered, $\{\,\}$ and $\{B_2\}$, and thus two intervals that span $C_1 \ldots A_2$: $\{\mathrm{A}, \mathrm{C}\}$ and $\{\mathrm{A}, \mathrm{B}, \mathrm{C}\}$ respectively. In total, three intervals are identified while the focus is on A_2.

Shifting the focus to the next point in the sweep, C_2, then opens up A_2 and B_2 as potential left-hand partners. The interval $\{\mathrm{A}, \mathrm{C}\}$ is first identified while A_2 is being evaluated as a left-hand partner; and then $\{\mathrm{B}, \mathrm{C}\}$ and $\{\mathrm{A}, \mathrm{B}, \mathrm{C}\}$ both emerge when it is B_2 that is being considered as a left-hand partner for C_2, covering $B_2 \ldots C_2$.

Lazy All Interval Querying, LPS-IF-A More generally, each time an item *rsym* at position *rpos* becomes the right-hand focus of the sweep, all right-most instances of other terms that lie between *rpos* and the prior occurrence of *rsym* can be used as the left-hand partners for intervals that terminate at *rpos*. The endpoints of that range define an interval over two terms; other intervals using the same range get added as a result of the need to include not just the endpoint items, but every combination of other terms that lies between them. Algorithm 3 captures these ideas.

The main loop in Algorithm 3 iterates over potential interval right-hand ends in variable *rpos*, drawing them out of a merging process on the terms' posting lists, as was also the case in Algorithm 1. The term identifier that matches this right-hand pivot value is stored as *rsym*. For any particular right-hand pivot point at *rpos*, there are at most $k - 1$ corresponding left-hand partners to be considered; moreover, to be interesting, the most recent occurrence of the candidate must fall between *rpos* and the previous appearance of *rsym*. That "last occurrence" location for each symbol in the query is maintained throughout the computation in the array *curr_occ*[*rsym*], with all values initialized to $-\infty$. If such an intervening symbol *lsym* is found, then its position *lpos* forms an interval with *rpos*, and a two-element interval can be emitted. Once a two-element interval has been identified, every distinct item that occurs between *lpos* and *rpos* can also be (or not be) a member of an interval with the same bounding endpoints. To accommodate that requirement, the set of in-between items is accumulated in S, and then every subsets $s \subseteq S$ is combined with the fixed endpoints *lpos* and *rpos* to create valid intervals. The whole process completes when all occurrences of all terms have been consumed out of the postings lists. This occurs when every *next_occ*[i] value is $+\infty$. Note that the pseudo-code shows all output information be-

Algorithm 3 Lazy Plane Sweep, All Intervals (LPS-IF-A)

Input: A query Q of k terms, and sorted position lists of all terms $\{P_1, P_2, \ldots, P_k\}$
Output: A set of intervals $\mathcal{I}$ for Q and all subqueries of Q

$\mathcal{I} \leftarrow \{\}$
for $i \leftarrow 1$ **to** k **do**
  curr_occ$[i] \leftarrow -\infty$
  next_occ$[i] \leftarrow$ FIRST(P_i)
end for
while $(\min_{1 \le i \le k}$ *next_occ*$[i]) < \infty$ **do**
  rsym $\leftarrow \arg\min_{1 \le i \le k}\{$*next_occ*$[i]\}$
  rpos $\leftarrow$ *next_occ*[*rsym*]
  for *lsym* $\leftarrow 1$ **to** k **do**
    if *curr_occ*[*lsym*] $>$ *curr_occ*[*rsym*] **then**
      // *lpos* can be LH end of intervals ending at *rpos*
      lpos $\leftarrow$ *curr_occ*[*lsym*]
      $S \leftarrow \{i \mid 1 \le i \le k$ **and** *lpos* $<$ *curr_occ*$[i]\}$
      for all subsets s of S, including the empty set **do**
        elems $\leftarrow s \cup \{$*lsym*, *rsym*$\}$
        $\mathcal{I} \leftarrow \mathcal{I} \cup \langle$*elems*, *lpos*...*rpos*$\rangle$
      end for
    end if
  end for
  curr_occ[*rsym*] $\leftarrow$ *rpos*
  next_occ[*rsym*] $\leftarrow$ NEXT(P_{rsym})
end while

ing accumulated into one stream $\mathcal{I}$, but that separate streams can also be generated if required, one per occurring subquery.

Eager All Interval Querying, EPS-CDF-A Algorithm 3 is a "lazy" implementation of the new approach; Algorithm 4 shows that the same idea can also be implemented in an eager manner using a condensed direct file. Now the primary loop iterates over potential left-hand positions, via variable *lpos*. For each value of *lpos* a forward scan is made using *rpos*, and *rsym*, building intervals of increasing size until *lsym* $=$ *rsym* and the bounding range for intervals whose left-hand end is at *lpos* has been found. The remainder of the process is similar between the two implementations – within the range *lpos*...*rpos*, all possible subsets of intervening items are used to form intervals. One important difference is that in Algorithm 4 terms are considered in position order, and the use of bitvector *seen* allows set S to be implemented as a stack when the pseudocode is translated into an actual program. On the other hand, in Algorithm 3, set S is rebuilt each time it is required by iterating over k possibilities, a factor that influences the relative execution times of the two methods in some situations.

Finally, note that while Algorithm 4 is presented assuming that an inverted file data structure is being used, it can also readily be executed starting with a direct file.

Correctness The discussion above already introduced the key notion that ensures that the two new algorithms are correct: *any optimal interval $l...r$ that ends with some symbol x cannot contain any other instances of x.* To see the truth of this claim, suppose that some v exists such that $l < v < r$ and (making use of the notation employed for the direct file) $D[v] = x$. Now consider the interval $l...r-1$. Only one symbol, $D[r] = x$, has been excluded relative to the original interval, $l...r$, and hence x is the only term that could possibly not also appear in $l...r-1$. But there is another location v with $l < v \le r-1$ for which $D[v] = x$. Hence, $l...r-1$ must be optimal for the same subset of terms as $l...r$,

Algorithm 4 Eager Plane Sweep, All Intervals (EPS-CDF-A)

Input: A query Q of k terms, and sorted position lists of all terms $\{P_1, P_2, \ldots, P_k\}$
Output: A set of intervals $\mathcal{I}$ for Q and all subqueries of Q

$(D', T_{Q,D}) \leftarrow$ BUILD_CDF$(\{P_1, P_2, \ldots, P_k\})$
$\mathcal{I} \leftarrow \{\}$
for $i \leftarrow 1$ **to** k **do**
  seen$[i] \leftarrow 0$
end for
for *lpos* $\leftarrow 1$ **to** $T_{Q,D} - 1$ **do**
  lsym $\leftarrow D'$[*lpos*]
  rpos $\leftarrow$ *lpos* $+ 1$ with *lsym* at left-hand end
  rsym $\leftarrow D'$[*rpos*]
  while *rpos* $\le T_{Q,D}$ **and** *lsym* $\ne$ *rsym* **do**
    if *seen*[*rsym*] $= 0$ **then**
      // *rpos* can be RH end of intervals starting at *lpos*
      $S \leftarrow \{i \mid 1 \le i \le k$ **and** *seen*$[i] = 1\}$
      for all subsets s of S, including the empty set **do**
        elems $\leftarrow s \cup \{$*lsym*, *rsym*$\}$
        $\mathcal{I} \leftarrow \mathcal{I} \cup \langle$*elems*, *lpos*...*rpos*$\rangle$
      end for
      seen[*rsym*] $= 1$
    end if
    rpos $\leftarrow$ *rpos* $+ 1$
    rsym $\leftarrow D'$[*rpos*]
  end while
  // reset *seen* to previous state
  for $i \leftarrow$ *lpos* **to** *rpos* **do**
    seen$[D'[i]] \leftarrow 0$
  end for
end for

contradicting the assumption that $l...r$ is optimal. A similar argument shows that if $D[l] = x$, and $l...r$ is optimal, then there cannot be any occurrence of x in $D[l+1...r]$.

In both Algorithms 3 and 4 all possible intervals that do not violate this requirement are identified, and all possible subsets of intervening items within such intervals are considered. Hence both algorithms correctly generate all optimal intervals for the query Q and all of its subqueries $Q' \subseteq Q$ for which $|Q'| \ge 2$.

Analysis We consider Algorithm 4 first. The main loop at step 6 iterates $T_{Q,D}$ times, where $T_{Q,D}$ is the size of the CDF D', and is equal to the sum of the lengths of the terms' postings lists. The inner loop at step 10 then iterates a variable number of times, hunting forwards through D' for the next occurrence of *lsym*, the symbol at D'[*lpos*]. In total, for each of the query terms, the loop at step 10 iterates at most $T_{Q,D}$ times, since each rightward scan starts at one occurrence of *lsym*, and stops when it reaches the next, forming a telescoping sum. In aggregate, step 11 is executed at most $k \cdot T_{Q,D}$ times over all values of *lpos*.

As already noted, S need not be re-evaluated each time step 13 is reached, because intervening symbols are added to it and never removed, allowing it to be stored compactly in a k-element array. Once S is formed, each iteration of the inner loop at step 14 takes $O(1)$ time, and adds one item to $\mathcal{I}$. Over all iterations of all loops, the total time spent in the loop commencing at step 14 is thus $O(|\mathcal{I}|)$.

The only other cost is at step 1; as already discussed, this requires $O(T_{Q,D} \log k)$ time if a heap is used to convert a set of postings lists to a CDF, or $O(N)$ time if the input is from a direct file. Note that in a pragmatic implementation, with k unlikely to be larger

Queries	Description
Set A	500 queries sampled from the 2007–2008 MQT logs, with exactly 100 queries for each length $k = 3, 5, 7, 8$, and 10
Set B	1,000 queries sampled from the 2007 MQT log, with a natural distribution of query lengths $3 \le k \le 12$

Table 1: Summary of the two sampled query sets generated from the TREC 2007–2008 Million Query Track topics and the TREC GOV2 document collection.

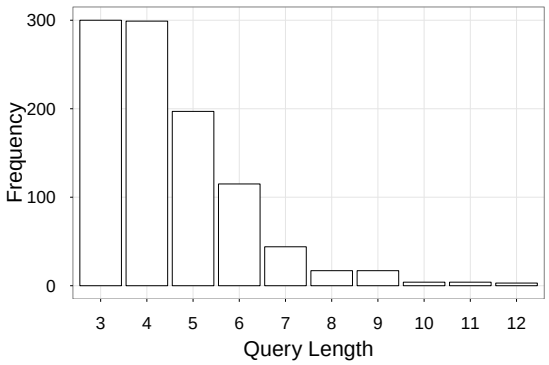

Figure 6: Length distribution for query Set B.

than around 10 or 15, an array (rather than a heap) is also a plausible choice for use during the IF to CDF conversion process, taking $O(T_{Q,D}k)$ time.

Summed over all components, the execution time of Algorithm 4 is thus $O(T_{Q,D}k + |\mathcal{I}|)$ when the input is provided as an inverted file, and $O(T_{Q,D}k + |\mathcal{I}| + N)$ if the input is a direct file.

Algorithm 3 is harder to analyze. The loop at step 6 iterates $T_{Q,D}$ times, with $O(T_{Q,D}k)$ or $O(T_{Q,D} \log k)$ time spent merging postings pointers, depending on the data structure used. For each iteration the loop at step 9 executes exactly k times and some number – possibly as many as all – of those executions makes it past the test at step 10, to execute the block of statements starting at step 11. At step 11, the process of constructing S requires $O(k)$ time, because *curr_occ* is not ordered. Finally, step 14 requires $O(1)$ time per interval that is generated, as was also the case with Algorithm 4. Putting all of these parts together, it is possible that on some inputs Algorithm 3 requires as much as $O(T_{Q,D}k^2)$ time. That worst-case bound could be reduced by maintaining *curr_occ* in sorted order via a permutation vector, paying an overhead cost to rearrange it every time it changes. As is shown below, for typical inputs the extra expense of that rearrangement is not warranted in terms of average case performance, but not doing it does indeed introduce the risk of more costly behavior on some queries.

5. EXPERIMENTS

Experimental Setup All experiments are performed on a machine using an Intel Xeon E5 CPU with 256 GB RAM running RHEL-v6.3 Linux, implemented in C++, and compiled using GCC 4.8.1 with –O2 optimization enabled, with execution times measured in milliseconds (msec) per document. All experiments use the TREC GOV2 collection. We also performed a similar series of experiments using the TREC ClueWeb09A data collection and obtained consistent results. Since our experiments focus on the set of top-ranked retrieved documents for each query, the results are influenced primarily by the nature of the documents retrieved, and agnostic to the overall collection composition.

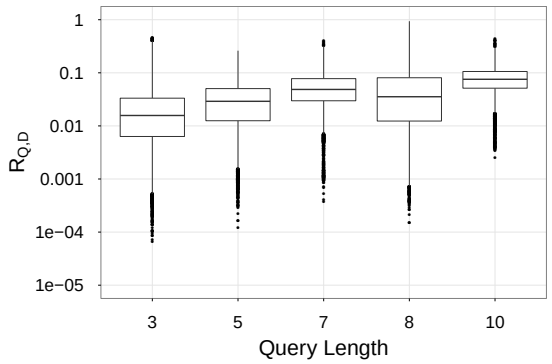

Figure 7: Distribution of $R_{Q,D}$ for Set A as a function of k.

The primary prerequisite to performing accurate comparisons of the algorithms we present is that a range of query lengths and document lengths be employed. To that end, two different query samples were created using the topics from the 2007–2008 Million Query Track (MQT) tasks. Table 1 summarizes the composition of the two query sets. Set A was developed in order to compare performance characteristics as a function of query length, with a focus on longer queries, and a minimum of 100 queries of each length. The MQT did not contain 100 queries of length 9. We also generated a second query set containing 1,000 randomly selected queries with $3 \le k \le 12$, in order to capture the query distribution in a real query stream. Longer queries exist in the MQT log, but did not emerge as part of our sample. We refer to this second set as Set B, and use it to compare performance characteristics when the sample distribution is representative of a real query stream, with short queries more probable than longer ones. The distribution of lengths in Set B is shown in Figure 6.

Each query in each set was then evaluated against GOV2 using Indri 4.7 with Krovetz stemming and a simple bag-of-words language model ranking, an out-of-the-box configuration, with stop words retained in the index. The 100 top-ranked documents for each query were retrieved, and used in conjunction with it as a query-specific pool of 100 documents against which the interval-finding algorithms might be plausibly applied in an operational setting, and against which execution times could be measured.

As discussed in Section 3, proximity evaluation depends on two key problem parameters – the length of the document, N, and the number of postings that must be processed, $T_{Q,D} = \sum_{i=1}^{k} |P_i|$. The *term density ratio* for a document D in respect of a query Q, defined as $R_{Q,D} = T_{Q,D}/N$, captures the relationship between these quantities. The direct file-based approaches are likely to be most useful when $T_{Q,D}$ is high, and many of the terms in a document are also in the query. The $R_{Q,D}$ distributions of the 50,000 documents extracted from GOV2 using Set A are shown in Figure 7. There is a slight trend for longer queries to be denser across the documents that they retrieve.

Single-Query Interval Finding We first compare the cost of direct files and inverted files. As previously discussed, EPS is most amenable to an inverted file input, and LPS to direct file input. But LPS can also be coupled with IF input – a condensed direct file can

k	LPS-DF-S		EPS-IF-S		LPS-IF-S	
	med.	max.	med.	max.	med.	max.
3	0.019	0.611	0.003	2.335	0.002	2.304
5	0.033	0.765	0.004	2.263	0.004	3.147
7	0.055	1.224	0.008	5.428	0.013	5.869
8	0.066	1.054	0.011	9.350	0.020	9.626
10	0.085	1.139	0.014	6.628	0.029	7.778

Table 2: Median and maximum time (msec/doc) spent finding optimal k-intervals using Set A.

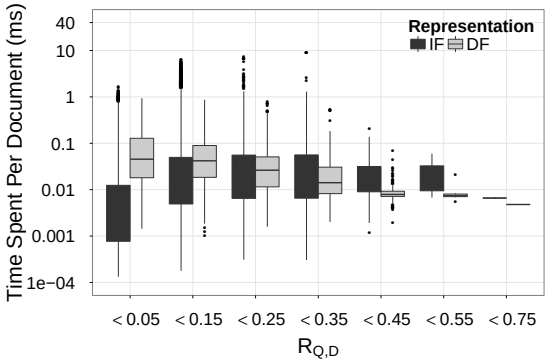

Figure 8: Time spent (msec/doc) constructing the condensed direct file, starting with an IF or a DF, using the Set A documents. There were no queries in the $0.55 \leq R_{Q,D} < 0.65$ range.

be implicitly constructed from position lists without affecting the asymptotic cost of the interval-finding task, allowing an LPS-IF-S combination to also be tested.

Table 2 lists optimal k-interval finding for the Set A queries for these three methods, with the -S suffix used to denote standard queries, without subquery intervals also being generated. The two IF-based methods exhibit similar behavior, but with EPS-IF-S consistently a little faster than LPS-IF-S. Both IF-based methods are faster than LPS-DF-S for most queries. However, LPS-DF-S is better in the worst case than both IF-based approaches, at all query lengths – situations in which $R_{Q,D}$ is high. When this happens, locality of access in the DF works better than iterating across several postings lists. Conversely, when $R_{Q,D}$ is small, the skips induced from the postings lists outperform a sequential scan of the direct file for most query-document combinations.

Consider two queries, as a concrete examples. In the three word query: "`mental health counselors`", all three terms tend to have a similar within-document term frequencies (around 200), and all of the optimal k-intervals appear close to each other. In this case, EPS-IF-S performs unnecessary comparisons and exchanges when trying to identify the rightmost position in the interval. However, in longer queries, such as "`what were the roles of african males in africa 1700s`" (recall that stop words are included) method LPS-IF-S performs poorly (7.64 msec). The same query on the same document is much faster with LPS-DF-S, taking 1.10 msec, with $R_{Q,D} = 0.16$.

Figure 8 captures these tensions, plotting the relative cost of constructing a condensed inverted file as a function of $R_{Q,D}$ for Set A. As expected, when $R_{Q,D}$ is small, it is much faster to start with an IF representation, whereas when $R_{Q,D}$ is large, the condensed file format can be created more efficiently starting with the DF format.

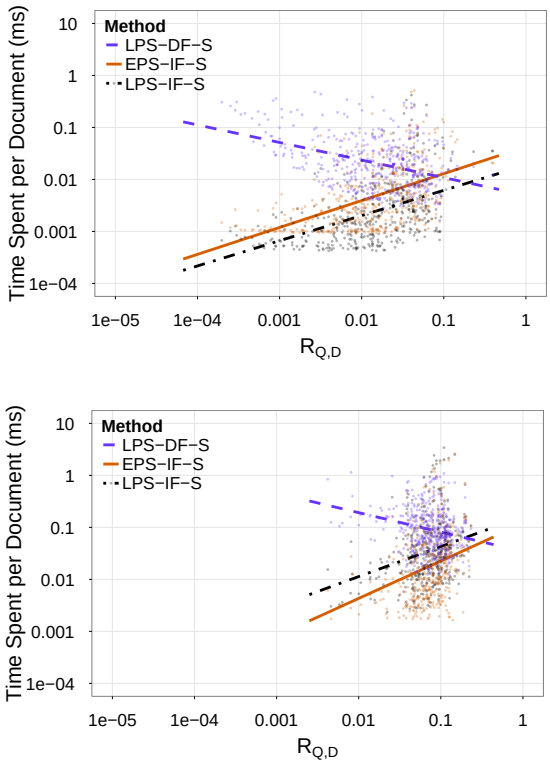

Figure 9: Time (msec/doc), as a function of $R_{Q,D}$ for Set A. The dashed blue regression line represents the LPS-DF-S method. The top figure shows results for $k = 3$ term queries, and the bottom one shows the results for $k = 10$ word queries. The plotted points represent a sample of 5% of the total data in each graph.

So, it seems conceivable that a query-sensitive mechanism might be devised to determine which input modality to use on a per-query basis if both formats were available at query time.

We now turn our attention to the performance of the interval-finding algorithms relative to $R_{Q,D}$ and query length. Figure 9 plots interval-finding time as a function of $R_{Q,D}$ for queries of length $k = 3$ (top) and $k = 10$ (bottom). The trend line for LPS-DF-S shows that query evaluation cost tends to decrease as $R_{Q,D}$ increases; whereas for EPS-IF-S and LPS-IF-S the trend is for cost to increase. For example, the three term query "`effingham county ga`" with $R_{Q,D} = 0.35$ performs poorly with both LPS-IF-S and EPS-IF-S, and is much more efficient with LPS-DF-S. The ten word query "`what was the population of the united states in 2000`" with $R_{Q,D} = 0.19$ exhibits similar behavior. Comparing the top panel to the bottom one, it can be seen that the "cloud" of points is shifted to the right in the $k = 10$ diagram, confirming that the LPS-DF-S approach is more likely to become superior as query lengths increase.

In general, our experiments for single-query interval finding show that an inverted file with position offsets is a better choice than a direct file representation. However, the worst case, while rare, is worth careful consideration. We will see shortly that the worst case costs are additive, and can have a significant impact when multiple interval computations are needed.

Computing All intervals for a Query The results in the previous experiments suggest that a direct file input representation is better

in a limited set of cases, but that the majority of the time, an inverted file starting point is most efficient. However, the potential for an additive worst case means we should proceed with care.

As shown in Section 4, the problem of enumerating optimal k-intervals for all possible subqueries can be solved in a single pass using either a Lazy or Eager evaluation strategy. In addition, there are a couple of other confounding factors. It is possible to create a condensed direct file representation once, and amortize the cost of the merge from an inverted file across multiple "single-interval query" computations.

In Table 3, two different approaches are considered when enumerating all of the possible $2^k - k - 1$ term patterns. The first approach is what systems currently do, compute each interval separately. Method LPS-IF-S* runs Algorithm 1 multiple times, starting each time with the postings lists, with the -S* suffix intended to indicate that the single-query process is run multiple times. Method LPS-CDF-S* takes a different approach. It first creates a CDF by merging the position offsets from the inverted files. The CDF is then processed $2^k - k - 1$ times, each pass making use of Algorithm 2 to compute the intervals for one of the possible subqueries. The contrast between these two algorithms is captured by comparing the median execution time, and the worst-case execution time. For most query-document combinations EPS-IF-S* is the best choice. However, the additive worst case is very bad, taking a full 3 seconds for one of the 10 word queries on a *single* document. The one-off cost of computing the CDF dramatically decreases the worst-case performance, but it can still be inefficient.

We can greatly boost the overall performance in both cases by computing all $2^k - k - 1$ possible subquery results in a single pass using either Lazy or Eager evaluation as described by Algorithms 3 and 4 in Section 4. The LPS-IF-A algorithm makes a single traversal using an inverted file as the initial input, with suffix -A indicating that all subquery intervals are found in a single pass. The EPS-CDF-A algorithm computes the CDF once by merging the position offsets from the inverted file. The CDF is then used to make a single pass using eager evaluation.

An important point to note about input representations with LPS and EPS based single pass algorithms is that either can benefit from computing the CDF. For a single-query interval, we demonstrated that the eager processing strategy is best suited for IF-structured input. However, this is not true when enumerating all intervals, because the corner case of having a huge gap between the current and next position for a term is offset by the fact that it is very likely that several of the subquery intervals will be formed even if the k-term interval has not formed yet. The extra work in resetting the left interval boundary often results in a subquery interval being emitted. In practical terms, this means that the performance profile for EPS-CDF-S* and LPS-CDF-S* is almost identical.

The key message in Table 3 is that performing the optimal k-interval enumeration process in a single pass is dramatically more efficient than either of the two multi-pass strategies, and, as expected, the gap becomes more pronounced as the query length increases. Although there are some small differences, the two -A plane-sweep approaches exhibit similar average performance when generating intervals for all subqueries. The key difference is again related to the worst-case. The tension between amortizing the creation of the CDF once and benefiting from locality of access, versus the cost of hitting a rare long document with many occurrences of the terms, separates the two approaches in the worst case.

Figure 10 shows the relative performance of multiple-pass and single-pass approaches. Note the use of a log scale. The upper panel shows the average performance for all 10,000 document-query combinations in Set B. Each boxplot summarizes the time

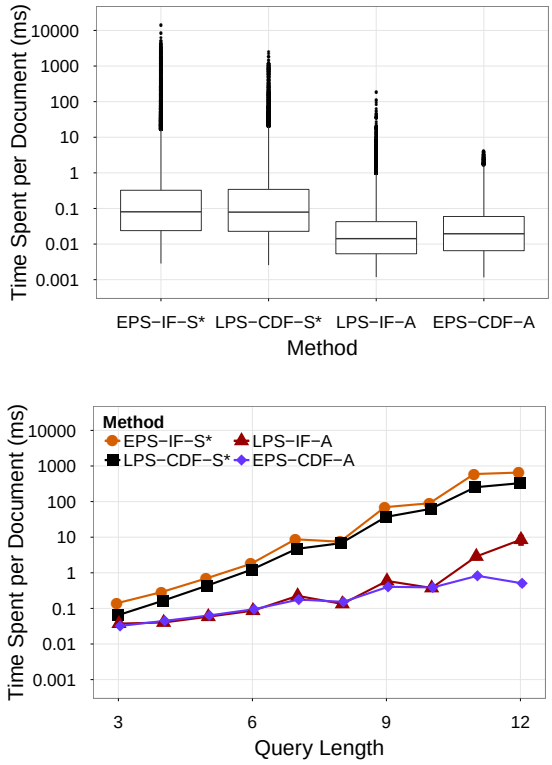

Figure 10: Time (msec/doc) to extract the optimal k-intervals for all subqueries using Set B. The top panel gives an overview of the per-query variation, incorporating all of the queries; the bottom panel shows the average time cost for each method broken down by query length. Note the log vertical scale on both graphs.

values as follows: the solid horizontal line indicates the median; the box shows the 25th and 75th percentiles; and the whiskers show the range, up to a maximum of 1.5 times the interquartile range, with outliers beyond this shown as separate points. In the overall performance comparison for all Set B queries, the LPS-IF-A algorithm appears to be slightly better than EPS-CDF-A, but a t-test shows that EPS-CDF-A is in fact significantly better with $p = 0.01$. This is largely due to the fact that the variance in query performance is more tightly bound for EPS-CDF-A than for LPS-IF-A. The bottom panel in Figure 10 shows the average (mean) processing cost in milliseconds per document as a function of query length. The two single-pass methods enjoy a near two-fold performance improvement for queries of length $k = 3$, and a more than 1000-fold improvement when $k = 12$.

The experiments we have described in this section leave absolutely no room for ambiguity. All-subqueries interval-finding is greatly accelerated by our two new methods, even when queries have as few as $k = 3$ terms in them.

6. CONCLUSION AND FUTURE WORK

We have explored two different questions. First, we considered two different approaches to efficiently compute an optimal k-interval based on input representation. We found that in most cases, starting with position offsets in an inverted file is more efficient than using a direct file. However, some query-document combinations can induce very bad behavior in k-interval computa-

k	EPS-IF-S*		LPS-CDF-S*		LPS-IF-A		EPS-CDF-A	
	med.	max.	med.	max.	med.	max.	med.	max.
3	0.011	7.300	0.011	1.307	0.006	2.821	0.007	0.986
5	0.089	37.770	0.104	8.757	0.018	3.758	0.024	1.790
7	0.778	344.800	1.113	59.060	0.060	7.037	0.071	3.282
8	2.104	1303.000	3.102	142.600	0.098	18.640	0.108	5.016
10	9.315	3397.000	17.160	801.600	0.175	46.350	0.191	9.222

Table 3: Median and maximum time (msec/doc) to find the optimal k-intervals for all subqueries for each query in Set A.

tions. By exploring the efficiency of the algorithms as a function of the term density ratio, we were able to isolate exactly when each evaluation strategy and input format is preferable.

In our second research question, we explored novel approaches to computing all term combination intervals in a single-pass. By exploiting a key insight in to the way intervals must be formed in subqueries, we showed how to significantly increase the performance of proximity computations. The performance gap between current approaches and our approach increases as query length increases, making it a viable tool in real search engines wishing to incorporate proximity computations into current ranking functions.

In future work, we will explore the impact on various interpretations of how intervals should be defined in IR ranking functions. Our current approach is more general than current FDM- and SDM-based ranking functions, which employed fixed-width window sizes, rather than optimal intervals. But note that the algorithms described in this paper can be adapted to also support fixed window sizes by counting only intervals that are less that a particular threshold.

Acknowledgment This work was supported by the Australian Research Council's *Discovery Projects* Scheme (DP140101587 and DP140103256). Shane Culpepper is the recipient of an Australian Research Council DECRA Research Fellowship (DE140100275).

References

[1] N. Asadi and J. Lin. Document vector representations for feature extraction in multi-stage document ranking. *Inf. Retr.*, 16(6):747–768, 2013.

[2] M. Bendersky, D. Metzler, and W. B. Croft. Parameterized concept weighting in verbose queries. In *Proc. SIGIR*, pages 605–614, 2011.

[3] A. Broschart and R. Schenkel. High-performance processing of text queries with tunable pruned term and term pair indexes. *ACM Trans. Information Systems*, 30(1):1–32, 2012.

[4] S. Büttcher, C. L. A. Clarke, and B. Lushman. Term proximity scoring for ad-hoc retrieval on very large text collections. In *Proc. SIGIR*, pages 621–622, 2006.

[5] S. Büttcher, C. L. A. Clarke, and G. V. Cormack. *Information Retrieval: Implementing and Evaluating Search Engines*. MIT Press, 2010.

[6] J. P. Callan, W. B. Croft, and S. M. Harding. The INQUERY retrieval system. In *Database and Expert Systems Applications*, pages 78–83. Springer, 1992.

[7] C. L. A. Clarke, G. Cormack, and F. J. Burkowski. Shortest substring ranking (multitext experiments for TREC-4). In *Proc. TREC*, pages 295–304, 1995.

[8] C. L. A. Clarke, G. V. Cormack, and E. A. Tudhope. Relevance ranking for one to three term queries. *Inf. Proc. & Man.*, 36(2):291–311, 2000.

[9] T. Elsayed, J. Lin, and D. Metzler. When close enough is good enough: approximate positional indexes for efficient ranked retrieval. In *Proc. CIKM*, pages 1993–1996, 2011.

[10] D. Hawking and P. Thistlewaite. Proximity operators-so near and yet so far. In *Proc. TREC*, pages 131–143, 1995.

[11] D. Hawking and P. Thistlewaite. Relevance weighting using distance between term occurrences. Technical Report TR-CS-96-08, Department of Computer Science, The Australian National University, 1996.

[12] S. Huston. *Indexing Proximity-Based Dependencies for Information Retrieval*. PhD thesis, University of Massachusetts, Amherst, 2014.

[13] S. Huston and W. B. Croft. A comparison of retrieval models using term dependencies. In *Proc. CIKM*, pages 111–120, 2014.

[14] D. Metzler. *Beyond Bags of Words: Effectively Modeling Dependence and Features in Information Retrieval*. PhD thesis, University of Massachusetts, Amherst, 2007.

[15] D. Metzler and W. B. Croft. Combining the language model and inference network approaches to retrieval. *Inf. Proc. & Man.*, 40(5): 735–750, 2004.

[16] D. Metzler and W. B. Croft. A Markov random field model for term dependencies. In *Proc. SIGIR*, pages 472–479, 2005.

[17] J. Peng, C. Macdonald, B. He, V. Plachouras, and I. Ounis. Incorporating term dependency in the DFR framework. In *Proc. SIGIR*, pages 843–844, 2007.

[18] K. Sadakane and H. Imai. Text retrieval by using k-word proximity search. In *Proc. Symp. Database Appl. Non-Trad. Envs.*, pages 183–188, 1999.

[19] R. Schenkel, A. Broschart, S. Hwang, M. Theobald, and G. Weikum. Efficient text proximity search. In *Proc. SPIRE*, pages 287–299, 2007.

[20] T. Tao and C. Zhai. An exploration of proximity measures in information retrieval. In *Proc. SIGIR*, pages 295–302, 2007.

[21] J. B. P. Vuurens and A. P. de Vries. Distance matters! Cumulative proximity expansions for ranking documents. *Inf. Retr.*, 17(4):380–406, 2014.

[22] H. Yan, S. Shi, F. Zhang, T. Suel, and J.-R. Wen. Efficient term proximity search with term-pair indexes. In *Proc. CIKM*, pages 1229–1238, 2010.

Author/s:
Lu, X;Moffat, A;Culpepper, JS

Title:
On the Cost of Extracting Proximity Features for Term-Dependency Models

Date:
2015

Citation:
Lu, X., Moffat, A. & Culpepper, J. S. (2015). On the Cost of Extracting Proximity Features for Term-Dependency Models. Bailey, J (Ed.) Moffat, A (Ed.) Aggarwal, CC (Ed.) Rijke, MD (Ed.) Kumar, R (Ed.) Murdock, V (Ed.) Sellis, T (Ed.) Yu, JX (Ed.) Proc. 24th ACM CIKM Int. Conf. on Information and Knowledge Management, 19-23-Oct-2015, pp.293-302. ACM. https://doi.org/10.1145/2806416.2806467.