Efficient Execution of Dependency Models

Samuel Huston
Center for Intelligent Information Retrieval
University of Massachusetts Amherst
Amherst, MA, 01002, USA
sjh@cs.umass.edu

W. Bruce Croft
Center for Intelligent Information Retrieval
University of Massachusetts Amherst
Amherst, MA, 01002, USA
croft@cs.umass.edu

ABSTRACT

Recently, several studies have investigated the efficient execution of bag-of-words (BOW) retrieval models for large document collections. Even though there is a large body of work that consistently shows that dependency models consistently outperform BOW models, none of these studies consider the efficient execution of dependency models. In this study we present initial investigations into the efficient execution of the sequential dependence model (SDM), a commonly used benchmark dependency model. We observe that versions of the Weak-And and Max-Score query processing models are efficient rank-k-safe query processing algorithms for this dependence model. Further, we observe that query processing time can be halved using a two-pass heuristic algorithm.

1. INTRODUCTION

Recently, there have been several studies that investigate the efficient execution of bag-of-words (BOW) retrieval models for large document collections [5; 6; 7; 12]. None of these studies consider the execution of dependency models, even though there is a large body of work demonstrating that dependency models can consistently outperform BOW models.

A simple solution is to index the required dependency features. A number of recent studies propose methods of fully or partially indexing dependency features [3; 8; 9]. These indexes allow the BOW-based query execution algorithms to operate for dependency models without modification. However, these indexes have either large space requirements, or discard a large fraction of dependency features, considerably reducing the effectiveness of the retrieval model for many queries.

Alternatively, positional indexes can be used to recreate a wide range of positional dependency features. However, processing positional data to identify instances of term dependencies considerably reduces query processing time. While this approach may not be appropriate for web-scale search solutions, it is appropriate for personal search on mobile devices, and for some enterprise settings where available disk and memory space is limited.

In this initial study, we investigate the adaptation of existing BOW algorithms for the execution of dependency models. We adapt two efficient document-at-a-time (DAAT) query processing algorithms to permit the rank-k-safe processing of dependency models using positional term indexes. We empirically compare the efficiency of versions of Max-Score [16], and Weak-And [2]. We also detail a heuristic two-pass approach that does not guarantee rank-safety.

We empirically evaluate the execution of the sequential dependence model (SDM) [10], a commonly used benchmark retrieval model, using each of these query processing algorithms. We compare each of the adapted algorithms to a naive baseline algorithm using three large-scale TREC collections.

2. RELATED WORK

A well studied alternative to DAAT algorithms are term-at-a-time (TAAT) algorithms. TAAT algorithms completely processes each query term, one at a time. This approach requires a significant amount of memory space, in the worst case, one partial score must be stored for each document in the collection. Efficient algorithms for early termination (ET) have been proposed [1; 11; 15]. These algorithms can be implemented on a variety of index organizations, including document-ordered, frequency-ordered, or impact-ordered indexes. However, TAAT algorithms prohibit the efficient evaluation of term dependency features using a positional index, as positional data for multiple terms is not available concurrently.

Two of the most commonly used DAAT algorithms are Max-Score and Weak-And. We discuss each of these algorithms in detail in Section 3. Some recent studies have extended these algorithms to use "Block-Max" indexes to improve the Max-Score [4; 5] and Weak-And [5; 6; 12] algorithms. Block-Max indexes provide access to the highest document frequency for each block of each posting list, allowing tighter estimation of upper bound scores for each block of each posting list. These studies have shown that the efficiency of BM25 [4; 5; 6; 12] and a variant of Query Likelihood [12] retrieval models may be improved using these statistics.

As term dependencies are computed from term position data for each document, the Block-Max values are not directly applicable to the dependency features. Even so, it may be possible for term-level Block-Max statistics to be used to tightly bound the contributions of term dependency features within a block. We leave the investigation of meth-

Algorithm NAÏVE

```
1: Input: ScoringIterator[] scorers
2: Output: ScoredDocument[] ranked-list
3: MinHeap topK
4: int prevDoc = −1
5: while true do
6:     int currDoc = ∞
7:     for ScoringIterator scorer : scorers do
8:         scorer.movePast(prevDoc)
9:         if not scorer.done() then
10:            currDoc = min(currDoc, scorer.doc())
11:        end if
12:    end for
13:    If all scorers are done return topK
14:    currScore = Σ_i scorers[i].score(currDoc)
15:    if topK.size() < k or currScore > topK.minScore()
       then
16:        topK.pop().push(currDoc, currScore)
17:    end if
18:    prevDoc = currDoc
19: end while
```

ods of using these statistics to estimate Block-Max statistics for term dependency statistics to future work.

Recent studies [7; 14] assert that the efficiency of information retrieval can be improved directly by storing large portions of the index in memory. This step reduces the cost of random-access disk seek times and allows for more efficient query processing. However, we note that this optimization is only appropriate for dedicated information retrieval systems. Using large amounts of RAM on personal computers or tablets to permit very rapid retrieval could have the undesirable effect of reducing the performance of other user applications. For this reason, we focus on disk-based retrieval in this study.

3. QUERY EXECUTION ALGORITHMS

The evaluation of a retrieval model for an input query is considered to be rank-equivalent to the weighted sum of a number of feature scorers. Each scorer is expected to be able to produce a score for each retrieval unit (document) in the collection. It is also expected to provide an upper bound of any document score, and an upper bound of the background score (a score produced for a zero-frequency document).

We assume access to a document-ordered positional index of terms. The reconstruction of pairs of terms into the required window instances is performed by intersecting pairs of posting lists. Note that this assumption means that term dependency collection statistics must be extracted from the pairs of posting lists prior to scoring any documents.

For SDM, the set of feature scorers include unigram, ordered window, and unordered window feature scorers. The posting list for each query term is read in as many as 5 feature scorers: a unigram and up to two ordered and unordered window scorers. So, there is a clear optimization available to all query execution algorithms – repeated reading and decoding of positional data can be avoided by sharing underlying readers.

3.1 Naïve

The NAÏVE algorithm is a baseline for each of the other algorithms tested in this initial study. This algorithm is a baseline that scores all documents that contain at least one of the query features. It does not attempt to minimize the number of documents decoded from the posting list data, or scored. Further, this algorithm allows scorers to share underlying posting list readers.

Algorithm MAX-SCORE

```
1: Input: ScoringIterator[] scorers
2: Output: ScoredDocument[] ranked-list
3: MinHeap topK
4: scorers = sort(scorers)
5: UBSum = Σ_i scorers[i].upper()
6: int prevDoc = −1
7: double threshold = −∞
8: while true do
9:     int currDoc = ∞
10:    for int i = 0; i < |scorers|; i+ = 1 do
11:        scorer.movePast(prevDoc)
12:        if not scorer.done() then
13:            allDone = false
14:            currDoc = min(currDoc, scorer.doc())
15:        end if
16:    end for
17:    if (all scorers are done) return topK.toArray()
18:    double currScore = UBSum
19:    int i = 0
20:    while i < |scorers| and currScore > threshold do
21:        currScore -= scorers[i].upper()
22:        currScore += scorers[i].score(currDoc)
23:        i += 1
24:    end while
25:    if i == |scorers| then
26:        if topK.size() < k) then
27:            topK.add(currDoc, currScore)
28:        else if currScore > topK.minScore() then
29:            topK.add(currDoc, currScore)
30:            threshold = topK.minScore()
31:        end if
32:    end if
33:    prevDoc = currDoc
34: end while
```

3.2 Max-Score

The MAX-SCORE algorithm was originally proposed by Turtle and Flood [16] for `tf-idf`-based scoring functions. We present an adaptation of this algorithm for the execution of arbitrary dependency retrieval models. The algorithm uses a threshold score t to determine when to discard the current document. To ensure rank-safety, t is set to the score of the k^{th} highest document.

In this algorithm, the score of a document is initially assumed to be the sum of the maximum possible contributions from each feature scorer ($UBSum$). The score is then adjusted as each feature is scored. Using this method, the running score can be directly compared to the threshold to determine if the document scoring can be terminated early. Similar to the NAÏVE algorithm, this algorithm allows scorers to share underlying posting list readers.

Unlike the original formulation of MAX-SCORE, this algorithm does not assume that the score of a zero-frequency feature is zero. By using the upper bound of the scorer as a reference point, retrieval models that return negative values,

Algorithm WEAK-AND

```
1: Input: ScoringIterator[] scorers
2: Output: ScoredDocument[] ranked-list
3: MinHeap topK
4: sort(scorers)
5: UBSum = sum_i scorers[i].upper()
6: ZeroUBSum = sum_i scorers[i].zeroUB()
7: int prevDoc = -1
8: double threshold = -infinity
9: while true do
10:     pivot = findPivot(scorers, ZeroUBSum,
       threshold)
11:     if (pivot < 0); return topK.toArray()
12:     if (scorer[pivot].done()); return topK.toArray()
13:     currDoc = scorers[pivot].doc()
14:     if (currDoc <= prevDoc
                or scorers[0].doc() < currDoc) then
15:         advanceScorer(scorers,
                max(prevDoc, (currDoc-1)))
16:     else
17:         currScore = 0
18:         for ScoringIterator scorer : scorers do
19:             currScore += scorer.score(currDoc)
20:         end for
21:         prevDoc = currDoc
22:         if topK.size() < k then
23:             topK.add(currDoc, currScore)
24:         else if currScore > topK.minScore() then
25:             topK.add(currDoc, currScore)
26:             threshold = topK.minScore()
27:         end if
28:     end if
29: end while
30: function findPivot(scorers[], ZeroUBSum, threshold)
31:     estScore = ZeroUBSum
32:     for (int i = 0; i<|scorers|; i+=1) do
33:         estScore -= scorers[i].zeroUB()
34:         estScore += scorers[i].upper()
35:         if (estScore > threshold) then return i
36:     end for
37:     return -1
38: end function
39: function advanceScorer(scorers[], prevDoc)
40:     select scorer i, s.t. scorer[i].doc() < prevDoc
41:     scorers[i].findCandidatePast(prevDoc)
42:     bubbleSortIter(scorers, i)      ▷ Only the scorer i
       needs to be reordered.
43: end function
```

such as log probabilities, can also be efficiently processed.

3.3 Weak-And

The WEAK-AND algorithm [2] operates by repeatedly selecting the next document to score using the upper bound of each scorer. Similar to the MAX-SCORE algorithm, the original algorithm was presented in reference to a strictly positive scoring function. Further, implicit in the original formulation of the WEAK-AND algorithm is the assumption that if a term does not occur in a document, then the score produced for the document is zero.

The adapted WEAK-AND algorithm, presented here, allows for arbitrary scoring functions. This is achieved through

Algorithm TWO-PASS

```
1: Input: ScoringIterator[] scorers
2: Output: ScoredDocument[] ranked-list
3: Separate unigram-scorers, and dependency-scorers
4: Using a rank-safe query execution algorithm, collect
   the top k_1 documents using the unigram-scorers
5: sort(docs_{k_1})
6: MinHeap topK
7: for (doc_i, score_i) in docs_{k_1} do
8:     for scorer in dependency-scorers do
9:         score_i += scorer.score(doc_i)
10:    end for
11:    if ((topK.size() < k_2)
               or (score_i > topK.minScore())) then
12:        topK.add(doc_i, score_i)
13:    end if
14: end for
```

the use of the upper bounds on background score (`zeroUB()`). These values are used to over-estimate the contribution of a scorer that does not contain the associated term or window. The FIND PIVOT function uses these bounds to select the next document to be scored.

The underlying readers associated with each scorer are moved in the ADVANCE SCORER function. This function also ensure that the scorers are kept in increasing order of document id. Unlike the previous algorithms, the WEAK-AND algorithm prohibits the use of shared readers.

A variant of this algorithm that shares underlying readers was tested. We found that the non-sharing version performed significantly faster for SDM across all collections. This is due to the added complexity of ensuring that the scorers remain in sorted order in the ADVANCE SCORER function.

3.4 Two-Pass Heuristic

The above algorithms are all considered rank-k-safe. Each of these algorithms guarantees that the actual top k documents are identified and returned, relative to the scoring function. We present here a simple non-rank-safe algorithm for dependency models. This algorithm is related to the cascade query execution model [17].

We start by separating the unigram feature scorers from the dependency feature scorers. A rank-k-safe algorithm is then used to identify the top k_1 documents, using the unigram features only. A second pass over the postings data finishes scoring these k_1 documents, and to collect the final k_2 documents. Where k_2 is the number of documents requested by the user, and $k_1 \geq k_2$.

For SDM, this process is equivalent to retrieving the top k_1 documents using Query Likelihood (QL) [13], and re-ranking these top documents using SDM.

While this algorithm reads and decodes each of the term posting lists twice, the first pass does not require positional data, and the second pass only scores a small fraction of the collection. For these reasons, we expect that this algorithm should be able to significantly reduce query processing time on larger collections.

4. INITIAL EXPERIMENTS

To empirically evaluate the efficiency of these algorithms we use the data from the TREC Million Query Tracks (MTQ). 500 short and 500 long queries are sampled from the 20,000

Table 1: Average query processing times for each algorithm, for long and short queries, for each collection. All reported query times are in seconds.

Query Set	Algorithm	GOV2	ClueWeb-09-B	ClueWeb-09-A
Short	Naïve	10·98	11·76	95·28
	Max-Score	7·45	7·83	45·87
	Weak-And	7·93	6·67	39·70
Short	Two-Pass-Naïve	4·25	4·82	30·42
	Two-Pass-Max-Score	4·14	4·36	27·00
	Two-Pass-Weak-And	3·72	3·80	24·57
Long	Naïve	43·44	62·73	527·7
	Max-Score	30·50	38·56	250·8
	Weak-And	28·00	36·40	261·1
Long	Two-Pass-Naïve	14·44	21·42	149·7
	Two-Pass-Max-Score	14·22	20·67	142·0
	Two-Pass-Weak-And	11·65	16·41	121·2

queries used in MTQ 2007/8, for the GOV2 collection. A further 500 short and 500 long queries from the 40,000 queries used in MTQ 2009, for the ClueWeb-09 collection. A short query is defined as containing between 2 and 4 words, and a long query is defined as containing between 5 and 13 words (inclusive).

For each of the query processing algorithms, we execute each batch of queries 5 times in randomized order. All experiments are executed in a single thread on a 3.0GHz Intel Xeon processor with 8G of RAM. 1,000 documents are retrieved for each query. The k_1 parameter for each of the Two-Pass algorithms is also set to 1000. This parameter setting maximizes the potential efficiency benefits of this heuristic algorithm.

The results of this initial experiment are shown in Table 1. These tables show that average time to execute a query on each collection using each algorithm. We observe that the Weak-And algorithm is the most efficient rank-k-safe algorithm in several settings. We also observe that the heuristic Two-Pass-Weak-And algorithm can further improve retrieval efficiency by between 38% and 53% for short queries, and between 53% and 58% for long queries.

5. FUTURE DIRECTIONS

Future work will include: the investigation of the trade-off between retrieval effectiveness and efficiency for the Two-Pass algorithms; the investigation of the relationship between the upper-bound scores, the upper bounds on the background scores and the actual scores for a variety of different retrieval functions; and an investigation of the relationship between the length of queries and the execution of the query.

Acknowledgments

This work was supported in part by the Center for Intelligent Information Retrieval and in part by NSF grant #CNS-0934322. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect those of the sponsor.

References

[1] Vo Ngoc Anh and Alistair Moffat. Pruned query evaluation using pre-computed impacts. In *Proceedings of the 29th ACM SIGIR*, pages 372–379, 2006.

[2] Andrei Z. Broder, David Carmel, Michael Herscovici, Aya Soffer, and Jason Zien. Efficient query evaluation using a two-level retrieval process. In *Proc. of the 12th CIKM*, pages 426–434, 2003.

[3] Andreas Broschart and Ralf Schenkel. High-performance processing of text queries with tunable pruned term and term pair indexes. *ACM Trans. Inf. Syst.*, 30(1):5:1–5:32, March 2012.

[4] Kaushik Chakrabarti, Surajit Chaudhuri, and Venkatesh Ganti. Interval-based pruning for top-k processing over compressed lists. In *Proc. of the IEEE 27th ICDE*, pages 709–720, 2011.

[5] Constantinos Dimopoulos, Sergey Nepomnyachiy, and Torsten Suel. Optimizing top-k document retrieval strategies for block-max indexes. In *Proc. of the 6th ACM WSDM*, pages 113–122, 2013.

[6] Shuai Ding and Torsten Suel. Faster top-k document retrieval using block-max indexes. In *Proc. of the 34th ACM SIGIR*, pages 993–1002, 2011.

[7] Marcus Fontoura, Vanja Josifovski, Jinhui Liu, Srihari Venkatesan, Xiangfei Zhu, and Jason Y. Zien. Evaluation strategies for top-k queries over memory-resident inverted indexes. *Proc. of the VLDB Endowment*, 4(12):1213–1224, 2011.

[8] Samuel Huston, Alistair Moffat, and W. Bruce Croft. Efficient indexing of repeated n-grams. In *Proc. of the 4th ACM WSDM*, pages 127–136. ACM, 2011.

[9] Samuel Huston, J. Shane Culpepper, and W. Bruce Croft. Sketch-based indexing of term dependency statistics. In *Proc. of the 21st ACM CKIM*, 2012.

[10] Donald Metzler and W. Bruce Croft. A markov random field model for term dependencies. In *Proc. of the 28th ACM SIGIR*, pages 472–479, 2005.

[11] Alistair Moffat and Justin Zobel. Self-indexing inverted files for fast text retrieval. *ACM Trans. Inf. Syst.*, 14(4):349–379, October 1996.

[12] Matthias Petri, Shane Culpepper, and Alistair Moffat. Faster top-k document retrieval using block-max indexes. In *Proc. of the 18th ADCS*, page To appear, 2013.

[13] Fei Song and W. Bruce Croft. A general language model for information retrieval. In *Proc. of the eighth CIKM*, pages 316–321, 1999.

[14] Trevor Strohman and W. Bruce Croft. Efficient document retrieval in main memory. In *Proc. of the 30th ACM SIGIR*, pages 175–182, 2007.

[15] Trevor Strohman, Howard Turtle, and W. Bruce Croft. Op-

timization strategies for complex queries. In *Proc. of the 28th ACM SIGIR*, pages 219–225, 2005.

[16] Howard Turtle and James Flood. Query evaluation: Strategies and optimizations. *Information Processing and Management*, 31(6):831 – 850, 1995.

[17] Lidan Wang, Jimmy Lin, and Donald Metzler. Learning to efficiently rank. In *Proc. of the 33rd ACM SIGIR*, pages 138–145, 2010.