Score-Safe Term Dependency Processing With Hybrid Indexes

Matthias Petri
RMIT University &
The University of Melbourne
Melbourne, Australia
matthias.petri@gmail.com

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

Score-safe index processing has received a great deal of attention over the last two decades. By pre-calculating maximum term impacts during indexing, the number of scoring operations can be minimized, and the top-k documents for a query can be located efficiently. However, these methods often ignore the importance of the effectiveness gains possible when using sequential dependency models. We present a hybrid approach which leverages score-safe processing and suffix-based self-indexing structures in order to provide efficient and effective top-k document retrieval.

Categories and Subject Descriptors

H.3.1 [**Information Storage and Retrieval**]: Content Analysis and Indexing—*indexing methods*; H.3.2 [**Information Storage and Retrieval**]: Information Storage—*file organization*; H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval—*query formulation, retrieval models, search process*; I.7.3 [**Document and Text Processing**]: Text Processing—*index generation*

General Terms

Text indexing; text compression; experimentation; performance

1. INTRODUCTION

Search engines rely on fast evaluation of ranking computations. Formulations based on bag-of-word queries and TF×IDF-type scoring mechanisms have been in use for several decades, and users are now accustomed to expressing their information needs via short keyword-based queries. One promising approach to improving the effectiveness of bag-of-words querying is term dependency modeling [12], which include statistics based on phrase combinations into the retrieval mechanism, to favor documents in which query terms appear consecutively, or near each other in unordered windows. The objective is to create a more favorable ordering than is achieved by a bag-of-words computation.

To process term-dependency queries, standard inverted index-based techniques can be augmented in two different ways. The first pre-defines queryable phrases at index construction time and includes information about them in the index, so that those phrases are automatically supported during query evaluation. The second approach includes word positions within the inverted index, so that arbitrary phrases can be identified during query evaluation. Compared to a document-level inverted index suited for bag-of-words retrieval, the first option has the disadvantage of either requiring a substantially enlarged inverted index or limiting the set of phrases that are supported; while the second option uses a smaller index, but needs additional processing when phrases are part of the query. Compromises between these two implementation options are also possible. Zobel and Moffat [20] give an overview of indexing and searching using inverted files.

The last decade has seen the emergence of other techniques, including self-indexing methods derived from suffix arrays and the Burrows-Wheeler transformation (BWT), following initial work by Muthukrishnan [13]. Until recently, these techniques were "single term" and based solely on occurrence frequency, reporting a ranked list of documents in decreasing frequency order in which any single fixed string of characters or words appears. That is, they report "top-k" for one-term queries, with "top" based on frequency alone. More recent work has explored the use of bag-of-word similarity computations, and stepped closer to the functionality that can be supported using document-level inverted indexes [3].

Our contribution. We implement term-dependency similarity models using suffix-based indexes, including combining single terms with multi-term phrases, and present an efficient safe-to-k approach for processing bag-of-word and phrase-based queries. Our method combines both pre-calculation and on-the-fly processing in order to achieve attractive time/space trade-offs, and demonstrates that multi-term queries can be processed using a suffix-based index more quickly than is possible using an inverted index alone.

2. PHRASE INDEXING AND QUERYING

Metzler and Croft [12] describe the use of query term dependencies to increase the effectiveness of similarity-based retrieval techniques. They make use of two operators, "ordered window" phrases, in which two or more of the query terms appear in a document in the sequence in which they appear in the query, and "unordered windows" in which two or more of the query terms appear within some specified distance of each other, but not necessarily adjacent, and not necessarily in the order in which they appear in the query. Metzler and Croft assign 80% of the final score of the document to a bag of words computation, 10% to ordered phrases, and 10% to unordered windows.

In the approach used here, scores are based on terms and a complete set of ordered phrases from the query. For example, the query "arthur conan doyle" is processed as a total of six components: the

SIGIR'14, July 6–11, 2014, Gold Coast, Queensland, Australia.

http://dx.doi.org/10.1145/2600428.2609469.

three terms, plus two 2-grams "arthur conan"; and "conan doyle" plus the 3-gram "arthur conan doyle". Including the t singletons, a query of t terms has $t(t+1)/2$ different components, each of which is scored across the collection as if it were a separate term.

Intersection-based phrase discovery. The simplest way of querying term m-grams is to store and process position offsets as part of an inverted index [20]. At query time, the various terms' postings lists are combined, and when terms co-occur in a document, a more detailed intersection operation is carried out to check for positional adjacencies. We call this the `intersect` method; for example, to create a query-time postings list for a phrase of m terms t_1 to t_m, the postings lists for the phrases $t_1 \dots t_{m-1}$ and $t_2 \dots t_m$ are intersected, with the base case supplied by the stored postings lists in the term-level inverted index.

Adding position offsets to the term-level inverted index of a typical document collection approximately doubles its size, a reasonable overhead; and when terms are infrequent, intersection of list components is also reasonably fast. But when long lists for common terms are involved, for example, in "the president of the united states", query processing can be costly.

Score-safe processing. A technique is *score-safe* if it guarantees that the top-k documents are the same as an exhaustive processing regime would generate. Score-safe methods include MaxScore [17], WAND [1], and BlockMax WAND (BMW) [4, 5]. All rely on a key insight: if the ranking metric is summative over terms, and if the maximum "contribution to any document" score of each term across the collection (or in the case of BMW, over a part of the collection) is known, then an evolving subset of the terms can be identified such that any document that might be added to the top k must contain at least one term in that set. Hence, attention can be restricted to documents containing one or more of those terms.

3. SUFFIX-BASED SELF-INDEXING

The FM-INDEX, introduced by Ferragina and Manzini [6], is a data structure used for pattern matching. Building on the *suffix array*, it also incorporates ideas embedded in the Burrows-Wheeler transform. A character-level FM-INDEX for a text can be stored in a fraction of the space occupied by the text itself, and provides pattern search and (with small overhead) random-access decoding from any location in the text. To build a word-level FM-INDEX, the input $\mathcal{T}$ is parsed into case-folded stemmed words exactly as for any other type of index, and retained as a sequence of word identifiers, $\mathcal{W}$, relative to a vocabulary. An end-of-document sentinel is inserted after each document in $\mathcal{W}$.

The sequence of integers is provided as input to a suffix-sorting algorithm, after which the FM-INDEX is constructed and stored. The FM-INDEX stores the symbols of $\mathcal{W}$ in a suffix-permuted ordering, $\mathcal{W}^{\text{BWT}}$, which allows efficient access, search and extraction of $\mathcal{W}$ using space equivalent to the compressed representation of the input. A third structure used during querying is the n-element *document array*, or *D-array*, which notes, for each element of $\mathcal{W}^{\text{BWT}}[i]$, the document number from which that word originated [13]. Each value $D[i]$ requires $\log d$ bits, where d is the number of documents in the collection; D takes $n \log d$ bits in total.

Single term query processing. Single term queries are processed as follows. First, the query is mapped to a sequence of integers, $\mathcal{Q}$, using the vocabulary. Using the FM-INDEX, a range $\mathcal{W}^{\text{BWT}}[sp..ep]$ is determined which corresponds to all suffixes in $\mathcal{W}$ prefixed by $\mathcal{Q}$. The corresponding range in the *D-array*, $D[sp..ep]$ now contains all document identifiers containing $\mathcal{Q}$.

The interval $D[sp..ep]$ corresponds to the total number of occurrences of $\mathcal{Q}$ in $\mathcal{W}$, which can be large; and hence processing $D[sp..ep]$ efficiently is critical to overall query performance. Various schemes have been proposed to identify the top-k most frequent document identifiers in $D[sp..ep]$. Culpepper et al. [2] show that if D is stored as a wavelet tree, the top-k documents can be found quickly in practice, but without a worst-case guarantee. Hon et al. [8] present a data structure that augments the D-array in order to provide worst-case bounds on both execution time and space. Their **HSV** mechanism pre-computes top-k result lists for a set of pre-determined $[sp..ep]$ intervals, guaranteeing that only relatively small ranges are processed exhaustively at query time. Recently, Konow and Navarro [10] presented a compact data structure which efficiently retrieves the top-k most frequent documents independently of the size of the range $[sp..ep]$.

Beyond single term top-k frequency retrieval. In the methods summarized so far, "top-k" has the semantics "return the k documents which contain the most occurrences of the pattern". These approaches cannot be easily adapted to process more complex queries. For example, the formulation of the **LMDS** similarity computation includes a range of factors, meaning that the top-k answer set to a query over two or more terms is not constrained to lie within the union of the terms' independent top-k answer sets.

Sadakane [15] described a support data structure which computes the number of unique document identifiers in $D[sp..ep]$ in constant time. Sadakane used this structure to support exhaustive processing over $[sp..ep]$, and hence calculate a simple TF×IDF based similarity metric. Culpepper et al. [3] adopt the **HSV** data structure [8] in combination with a wavelet tree-based representation [2] to compute similarity scores for multi-term queries. Their scheme retrieves a fixed number of most frequent document identifiers for each query term, and combines them to retrieve a ranked document listing, an approach that gives competitive runtime performance for $k \leq 100$, but is not score-safe.

4. A BLEND OF TECHNIQUES

We build on the previous work, and explore ways of implementing more complex score-safe query semantics – in particular, the sequential dependency model outlined earlier. The approach described shortly is a hybrid, consisting of a pruned suffix tree of document-level postings lists to efficiently handle large intervals, and fast sequential exhaustive processing of smaller sections to create document-level postings lists on-the-fly.

Pruned suffix tree. The set of intervals in the D-array that can be generated by the FM-INDEX correspond to the internal nodes of a suffix tree over $\mathcal{T}$. The **HSV** mechanism [8] attaches partial precomputed postings lists to nodes within that suffix tree, in order to compute the top-k (by frequency) values at any suffix tree node using a controlled amount of time. The length of each posting list is determined by the size of the corresponding $[sp..ep]$ interval, the number of sampled points within it, and its location in the suffix tree over $\mathcal{W}$; and is strategically balanced so as to achieve attractive time and space bounds.

We return to a simpler scheme which embeds several observations that apply to similarity metrics. Instead of the differing-length lists of the **HSV** structure, we store full postings lists every node in the suffix tree that corresponds to an interval wider than T_{min}, and compute postings lists on-the-fly for $[sp..ep]$ intervals that are

smaller. That is, we store a *pruned suffix tree* (PST) of full postings lists, for a subset of the intervals. A second threshold $T_{\max}$ is also defined – postings lists are not stored for intervals greater than $T_{\max} = d$ (the number of documents in the collection), since such terms have negligible bearing on the **LMDS** similarity computation.

Query evaluation. When a query identifies a small $[sp..ep]$ interval, a posting list is computed from the D-array via a sequential cache-friendly process in $\mathcal{O}(ep - sp)$ time. The longer the phrase and more precise the interval, the faster this computation is.

The thresholds $T_{\min}$ and $T_{\max}$ allow index space and execution cost to be traded – the broader the band between the thresholds, the larger the index, and the faster that query evaluation can be carried out. Our query processing scheme is summarized as follows: (1) determine for each query component (including all phrase components) the corresponding $[sp..ep]$ interval using the vocabulary and the FM-INDEX; then either (2a) retrieve the posting list from the PST if $T_{\min} \le ep - sp \le T_{\max}$, or (2b) create the postings list on the fly by processing $D[sp..ep]$ if $ep - sp < T_{\min}$; then (3) apply DAAT or MaxScore or WAND or BMW to the complete set of postings lists to identify the top-k documents.

5. EXPERIMENTS

We took the TREC GOV2 collection, containing 426 GB of web pages crawled from the `.gov` domain, and applied the Boilerpipe software package[1] to generate a plain text version occupying90 GB. A total of 150 test topics and corresponding relevance judgments are available for this collection (topics 701–850). Algorithms were implemented using C++ and compiled with gcc 4.8.1 with –O3 optimizations; and experiments were run on a 24 core Intel Xeon E5-2630 running at 2.3 GHz using 256 GB of RAM. All efficiency runs are reported as the mean or median of 5 consecutive runs of a sequence of at least 100 queries for a given query length, and correspond to fully in-memory evaluation, under optimal conditions. All results shown in this short paper are for identification of the top $k = 100$ documents, and with $T_{\min} = 64{,}000$. Other combinations of parameters will be explored in a full paper.

Each postings list is stored and compressed in blocks of 128 entries using the FastPFOR library [11]. All methods can take advantage of block representatives to allow faster skipping of list entries. Position offsets are stored using the succinct storage scheme of Vigna [18]. The FM-INDEX, the position offset representation, and the PST are implemented using the sdsl library [7].

Baseline performance. Table 1 shows the effectiveness of NewSys relative to Indri[2] and Atire [16]. The FDM model uses both phrase expansion and unordered-window proximity. Our approach uses only the phrase expansion component, and provides a compromise between the lesser effectiveness achievable using the efficient BOW approach, and the more complex FDM mechanism.

Figure 1 shows the effect that length has on query execution time for score-safe processing methods, using queries drawn randomly from the TREC million query set. The times reported in Figure 1 do not include phrase postings list creation cost, and cover only raw posting list evaluation cost after all required lists have been generated. The sequence of improvements arising from score-safe heuristics – WAND and BlockMax WAND (BMW) – is clear.

Improvements. The bars in Figure 2 show the relative cost of different approaches to on-the-fly construction for index lists that are not in the pruned index, normalized (at each query length) to the cost of processing the query using the exhaustive DAAT approach. The horizontal lines show the relative costs of different processing strategies, as already shown (in absolute terms) in Figure 1, normalized against the same reference costs. That is, the bars correspond to different ways of generating a full set of component postings lists, and the horizontal lines to different ways of then computing (the same top-k) document scores from those postings lists. A retrieval mechanism requires both steps. In Figure 2 the method labeled `intersect` is the baseline intersection-based approach; `darray` is our hybrid approach with phrases below the threshold $T_{\min}$ generated on the fly from the stored D-array; and `intersect-ppst` is described shortly.

Figure 2 shows very clearly that sequential processing of the D-array is far more efficient than generating postings lists from an inverted index via iterated intersections. The longer the query, the greater the advantage. Also worth noting is that iterative intersection dominates the similarity computation: `intersect`+BMW is only marginally faster than `intersect`+DAAT. Using the proposed `darray` approach the full benefits of BMW can be realized.

Table 2 lists timings for the queries shown in Figure 2. These times should be regarded as being indicative rather than precise: Indri and Atire were run with standard out-of-the-box configurations and each query was measured after a pre-execution to bring the required data in to memory; but the timings might still not be exactly like-for-like. Even so, the new approach provides clear benefits.

System	Factors	MAP
NewSys	BOW	0.278
Atire	BOW	0.279
Indri	BOW	0.280
NewSys	Phrase	0.298†
Indri	FDM	0.317†
Best Known	Unknown	0.334†

Table 1: Effectiveness (MAP) on GOV2 for topics 701–850. The "Best Known" system is `uwmtFadTPFB` from the TREC 2006 Terabyte Track. All systems use Krovetz stemming, the LMDS similarity formulation with $\mu = 2500$, and Boilerpipe preprocessing. A † represents $p < 0.05$ using a paired t-test against the BOW runs.

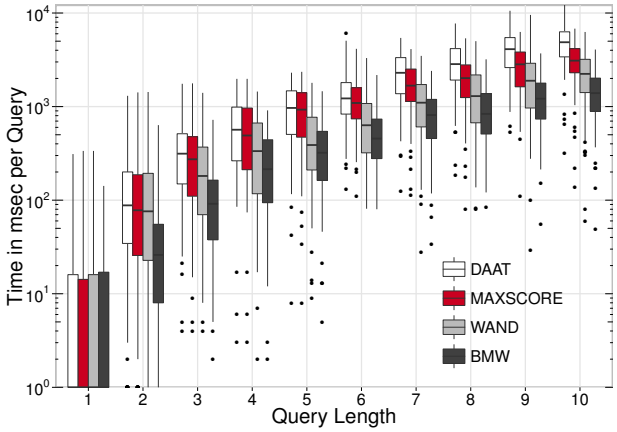

Figure 1: Query time distribution for different processing approaches using GOV2 and query lengths from 1 to 10. Note the logarithmic vertical scale.

[1] `https://code.google.com/p/boilerpipe/`
[2] `http://www.lemurproject.org/indri.php`

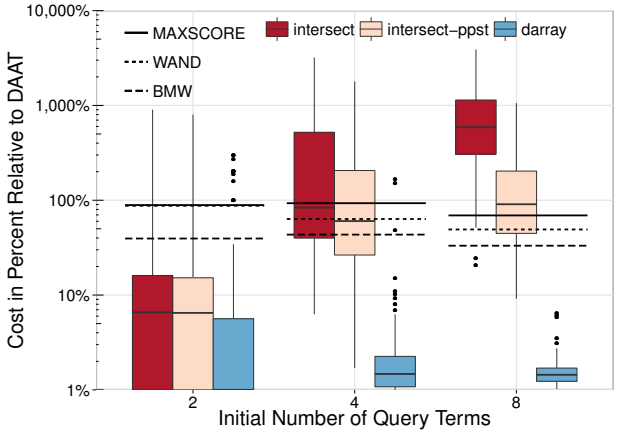

Figure 2: Relative query costs, normalized so that the list processing time using the DAAT strategy is rated at 100.0%.

System	Factors	$t = 2$	$t = 4$	$t = 8$
Atire	BOW	501	1814	5920
Indri	BOW	270	1580	9844
NewSys	Phrase	57	292	1084

Table 2: Average query processing time (milliseconds) on GOV2. The NewSys implementation uses the `darray`+BMW combination.

The performance advantages of the `darray` method come at a cost. It uses a total of 97 GB for the FM-INDEX, the **PST**, and the D-array; whereas the `intersect` method requires only 33 GB for a positional inverted index over terms. An obvious question is: can this space difference be used instead to index a subset of important phrases? Several authors have suggested just such a trade-off approach in which a subset of important phrases is added to the index (see Section 6). The `intersect-ppst` method shown in Figure 2 offers a pragmatic compromise between using position offsets for each term to generate the phrase components, and indexing all possible phrases. Since the PST already captures all possible phrases of frequency between T_{max} and T_{min}, we can add positional offsets to those inverted lists, covering the intermediate lists needed to construct postings lists for long phrases. Any remaining lists required are still generated on demand using iterated intersection. With the same value of T_{min}, this approach requires 101 GB, and $t = 5$ term (15 component) queries take an average of 1,700 millisec, compared to 400 millisec for the `darray`, and 5,800 millisec for `intersect`. That is, for our chosen query semantics the `darray` offers superior performance even when compared to a similar-sized inverted index that includes positional postings lists for all phrases (of any length) that occur more than T_{min} times.

6. RELATED WORK

Williams et al. [19] propose a modified inverted index, in which each posting list includes a record of what the next terms in the document are. Williams et al. also index common phrases as a single term within an inverted index in order to increase the efficiency of n-gram processing. Other researchers have also explored indexing common phrases, for example, Huston et al. [9] and Ozcan et al. [14]. These approaches provide efficient querying, but also have disadvantages: the phrase length is usually fixed at indexing time; index sizes can grow quickly; and infrequent phrases might not be recorded in the index at all.

7. CONCLUSION

We have explored algorithmic components that can be used to support efficient phrase querying as a contributing factor in document similarity ranking using term dependency models. While the complete system is still not as effective as the best available systems (which make use of non-consecutive term pairs, and/or query expansion), the efficiency advantages of the BWT-based index are appealing. The BWT-based systems can also support arbitrarily long exact match phrase queries with no additional space costs beyond the fixed D array overhead. Indeed, as the phrases get longer, the processing time gets faster, since the corresponding $[sp..ep]$ interval gets smaller.

Acknowledgments. This work was supported in part by the Australian Research Council (DP110101743). Shane Culpepper is the recipient of an ARC DECRA Research Fellowship (DE140100275). Simon Gog designed and implemented the pruned suffix tree used in the experiments and integrated it into the IR framework.

References

[1] A. Z. Broder, D. Carmel, H. Herscovici, A. Soffer, and J. Zien. Efficient query evaluation using a two-level retrieval process. In *Proc. CIKM*, pages 426–434, 2003.

[2] J. S. Culpepper, G. Navarro, S. J. Puglisi, and A. Turpin. Top-k ranked document search in general text databases. In *Proc. ESA*, pages 194–205, 2010.

[3] J. S. Culpepper, M. Petri, and F. Scholer. Efficient in-memory top-k document retrieval. In *Proc. SIGIR*, pages 225–234, 2012.

[4] C. Dimopoulos, S. Nepomnyachiy, and T. Suel. Optimizing top-k document retrieval strategies for block-max indexes. In *Proc. WSDM*, pages 113–122, 2013.

[5] S. Ding and T. Suel. Faster top-k retrieval using block-max indexes. In *Proc. SIGIR*, pages 993–1002, 2011.

[6] P. Ferragina and G. Manzini. Indexing compressed text. *J. ACM*, 52 (4):552–581, 2005.

[7] S. Gog, T. Beller, A. Moffat, and M. Petri. From theory to practice: Plug and play with succinct data structures. In *Proc. SEA*, 2014.

[8] W.-K. Hon, R. Shah, and J. S. Vitter. Space-efficient framework for top-k string retrieval problems. In *Proc. FOCS*, pages 713–722, 2009.

[9] S. Huston, J. S Culpepper, and W. B. Croft. Sketch-based indexing of n-words. In *Proc. CIKM*, pages 1864–1868, 2012.

[10] R. Konow and G. Navarro. Faster compact top-k document retrieval. In *Proc. DCC*, pages 5–17, 2013.

[11] D. Lemire and L. Boytsov. Decoding billions of integers per second through vectorization. *Soft. Prac. & Exp.*, 2014. To appear.

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

[13] S. Muthukrishnan. Efficient algorithms for document retrieval problems. In *Proc. SODA*, pages 657–666, 2002.

[14] R. Ozcan, I. S. Altingovde, B. B. Cambazoglu, F. P. Junqueira, and O. Ulusoy. A five-level static cache architecture for web search engines. *Inf. Proc. & Man.*, 48(5):828–840, 2011.

[15] K. Sadakane. Succinct data structures for flexible text retrieval systems. *J. Discr. Alg.*, 5(1):12–22, 2007.

[16] A. Trotman, X.-F. Jia, and M. Crane. Towards an efficient and effective search engine. In *Wkshp. Open Source IR*, pages 40–47, 2012.

[17] H. Turtle and J. Flood. Query evaluation: strategies and optimizations. *Inf. Proc. & Man.*, 31(6):831–850, 1995.

[18] S. Vigna. Quasi-succinct indices. In *Proc. WSDM*, pages 83–92, 2013.

[19] H. E. Williams, J. Zobel, and P. Anderson. Fast phrase querying with combined indexes. *ACM TOIS*, 22(4):573–594, 2004.

[20] J. Zobel and A. Moffat. Inverted files for text search engines. *ACM Comp. Surv.*, 38(2):6–1 – 6–56, 2006.