On the Cost of Phrase-Based Ranking

Matthias Petri Alistair Moffat
Department of Computing and Information Systems
The University of Melbourne
matthias.petri, ammoffat@unimelb.edu.au

ABSTRACT

Effective postings list compression techniques, and the efficiency of postings list processing schemes such as WAND, have significantly improved the practical performance of ranked document retrieval using inverted indexes. Recently, suffix array-based index structures have been proposed as a complementary tool, to support phrase searching. The relative merits of these alternative approaches to ranked querying using phrase components are, however, unclear. Here we provide: (1) an overview of existing phrase indexing techniques; (2) a description of how to incorporate recent advances in list compression and processing; and (3) an empirical evaluation of state-of-the-art suffix-array and inverted file-based phrase retrieval indexes using a standard IR test collection.

Categories and Subject Descriptors

H.3.1 [**Information Storage and Retrieval**]: content analysis and indexing—*indexing methods*; H.3.4 [**Information Storage and Retrieval**]: systems and software—*performance evaluation*.

1. INTRODUCTION

Postings list processing schemes such as WAND [2] have greatly decreased the cost of similarity computations for bag-of-words retrieval; and postings list storage schemes such as Elias-Fano codes have also reduced the cost of manipulating document-level postings lists [12]. But phrases are sometimes a further part of similarity computations such as BM25 and language models, and including them in similarity computations requires additional information, which must be either explicitly stored in the index, or computed at query time. A range of storage schemes have been proposed that provide different query time, storage space, and retrieval effectiveness trade-offs [4, 11, 19]. Of especial interest is the relative performance of traditional inverted index-based schemes and suffix array-based indexes. The latter have received significant recent attention because they support arbitrary phrase searching.

We compare storage schemes that support phrases as part of ranked queries, when evaluated via list processing schemes such as WAND. Specifically, we define a collection $\mathcal{D}$ of N documents $\mathcal{D} = d_0, \ldots, d_{N-1}$. Each document consists of a non-empty string of symbols drawn from an alphabet Σ of size σ consisting of the parsed word tokens of the collection. We additionally define a sequence $\mathcal{C} = d_0\$d_1\$d_2 \ldots d_{N-1}\$$ consisting of the concatenation of the documents in $\mathcal{D}$, separated by a symbol $\$ \notin \Sigma$. The length of $\mathcal{C}$ is given by n. A phrase $\mathcal{P}$ consists of a pattern $\mathcal{P}[0 \ldots m-1]$ drawn from Σ, where $m \geq 2$. The frequency of $\mathcal{P}$ in document d_i is denoted by $f_{i,\mathcal{P}}$. We seek to address the following requirement:

Sorted Document Listing with Frequencies – Preprocess a document collection $\mathcal{D}$ such that, given an arbitrary phrase $\mathcal{P}$, an increasing sequence of $N_\mathcal{P}$ document identifiers $\langle i, \ldots, j\rangle$ can be efficiently computed, together with the frequencies $f_{i,\mathcal{P}}, \ldots, f_{j,\mathcal{P}}$, where $d_k \in \mathcal{D}$ and $f_{k,\mathcal{P}} > 0$ for each $k \in \langle i, \ldots, j\rangle$.

The sorted document listing can then be used as a component in a similarity computation using algorithms such as WAND, exactly as if it had been drawn directly from a pre-computed phrase index.

2. PHRASE INDEXING SCHEMES

Five broad categories of phrase indexing scheme have been proposed, with combinations providing other trade-offs.

Positional. A list of term positions is stored, relative to the start of each document (hierarchical), or as absolute positions within $\mathcal{D}$ [17]. The relative representation requires less space, but at the cost of increased run-time performance, since intersection has to be performed both at the document and at the positional level. To answer phrase queries using absolute positions, the positional lists of the terms are intersected using an algorithm such as SvS [4], then converted to an increasing list of document identifiers and frequencies by mapping each position to a corresponding document number.

Pre-computation. Instead of performing list intersection at query time, the final set of $\langle i, f_{i,P}\rangle$ pairs can be stored in the index and accessed when needed by queries. Storage limits mean that pre-computing postings lists for all phrases is impossible, and techniques have been explored to choose lists to be computed, including analyzing query logs [3, 19] and using collection statistics [13]. Indexing only a subset of the phrases implies that either other ways of creating lists at query time must be provided too, or that retrieval effectiveness must be sacrificed.

Suffix-Array. Suffix arrays and suffix trees have primarily been used to support regular pattern searches over a continuous text; but suffix array-based structures can also be used for the sorted document listing problem [10, 11]. All suffix array-based indexes are structured as follows. First, a suffix array $SA[0 \ldots n-1]$ over $\mathcal{C}$ is constructed. Then an array $DA[0 \ldots n-1]$ is added, to store the document number in which each suffix appears. To compute a document listing, the interval $SA[sp, ep]$ containing all suffixes prefixed by $\mathcal{P}$ is determined, by searching the suffix array. The unique

SIGIR'15, August 09-13, 2015, Santiago, Chile.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-3621-5/15/08 ...$15.00.
http://dx.doi.org/10.1145/2766462.2767769.

document identifiers and their frequencies in *DA*$[sp, ep]$ are then determined, using one of several possible methods. In this work we compare three suffix array-based indexes that provide a range of time-space trade-offs.

As a suffix array baseline we use the SORT [6] index. It replaces *SA* by a compressed suffix array (CSA) [14], which reduces the space required; and stores *DA* as an array using $n\lceil\log_2 N\rceil$ bits. To determine document identifiers and frequencies for a range *DA*$[sp, ep]$, it is copied to a temporary vector and sorted using a standard integer sorting algorithm. One additional pass produces the list of all $N_{\mathcal{P}}$ pairs $\langle i, f_{i,\mathcal{P}}\rangle$ matching $\mathcal{P}$. The second option, Sadakane's (SADA) index [15] also uses a CSA to find $\langle sp, ep\rangle$. But instead of storing *DA* explicitly, it is emulated using a bitvector of length n that marks the document boundaries. Two range minimum query (RMQ) data structures are used, and each unique document identifier in the range is processed only once. In addition, for each document d_i a separate CSA is stored, to determine $f_{i,\mathcal{P}}$. The third variant (WT) stores *DA* using a wavelet tree [5]. After determining *DA*$[sp, ep]$, the wavelet tree supports retrieval of the $N_{\mathcal{P}}$ pairs $\langle i, f_{i,\mathcal{P}}\rangle$ in sorted order in $O(N_{\mathcal{P}} \log \sigma)$ time [16].

Navarro [11] provides details of these mechanisms.

Next Word Indexing. Instead of storing position lists for all unique terms in Σ, positions for all unique bi-grams in $\mathcal{C}$ can be stored [19]. This requires additional space, but intersection of long phrases can be performed efficiently, as the bi-gram lists are shorter than the corresponding term lists. Williams et al. [19] explore additional trade-offs where only some bi-gram lists are stored explicitly, based on common phrases identified in a query log.

Document Surrogates. Direct sequential search for phrases can be carried out in parsed documents if a small number of candidates is identified via an initial document-level search process. These methods make use of the "direct" file that is maintained by some search systems. Other indexes tailored towards proximity queries also cannot efficiently be used to answer phrase queries [1, 20].

3. LISTS AND INTERSECTION

Indexes that make use of postings lists store sorted sequences of integers, both as part of the underlying document level inverted index, and as part of the positional index. Compression techniques such as OptPforDelta [4] and representations based on Elias-Fano codes [12, 18] allow compact storage of sorted lists, and support fast execution. One of our purposes in this work is to explore the benefits of these new structures in the context of phrase processing.

Packed Binary. One of the most effective encoding schemes for postings lists is OptPforDelta [21], which compresses blocks of integers using a fixed width b, storing values larger than $2^b - 1$ as exceptions using a secondary codec. The underlying structure of the code, and the availability of SIMD instructions to assist the process [9], means that implementations can be very fast, and well-suited to hierarchical indexes. On the other hand, absolute positional indexes require encoding schemes which support large integers, and while all of the methods described can be adapted to support 64-bit values, underlying SIMD instructions may not be available, and decoding may be slower. Similarly, compression schemes tuned for positional information may be slower than document level compression codecs [20].

Bitvector Hybrids. Kane and Tompa [7] combine bitvector representations and standard compression techniques. Each postings list in the index is either stored completely as a bitvector; partially using a bitvector for the first part and then a compressed representation for the tail; or fully compressed. But bitvectors are only viable if the set being represented is dense across the domain. For example, a bitvector representing a domain of $N \approx 25$ million documents requires ≈ 3 MiB. The domain is much sparser when storing positional information; and for a collection containing $n \approx 23$ billion words, a bitvector would be 2.8 GiB. That is, the methods of Kane and Tompa (and the earlier work of Culpepper and Moffat [4]) are not suited to storing positional information.

Elias-Fano Codes. The Elias-Fano (EF) representation for increasing sequences has been rediscovered recently and applied to postings lists [18]. Elias-Fano codes provide a principled way of including bitvectors into postings list representations. Each number is split up into low and high parts. The high parts of the numbers in a sequence are stored as unary encoded differences in a bitvector; the low parts in binary using a fixed number of bits. Ottaviano and Venturini [12] show that a block-partitioned two-level EF representation can compete in both space and time with other forms of posting list compression. Like some other coding methods, the partitioned EF scheme exploits locality within postings lists. Document reordering, typically based on source URLs, produces clusters of similar documents and hence non-uniform postings lists, and can improve compression effectiveness and list processing speed by up to 40% [7]. However, these clusterings schemes are not applicable to improving compression for lists of term positions [20].

Intersection. Conjunctive bag-of-words queries and phrase searching using positional indexes both rely on list intersection, with skip information or bitvectors used to improve intersection speed [4, 7]. Elias-Fano codes directly include bitvectors, and Vigna [18] demonstrates how the bitvector representing the high parts of the sequence can be used as part of the compressed representation of the sequence, as well as an auxiliary structure to support fast skipping. Ottaviano and Venturini [12] show that Boolean conjunctive queries using EF codes outperform OptPforDelta schemes, support 64-bit integers, and are competitive to other representations optimized to support fast list intersection.

4. EXPERIMENTS

Dataset and Methodology. We use the standard GOV2 test collection of the TREC 2004 Terabyte Track, stored in URL-sort order. To ensure reproducibility we extract the integer token sequence $\mathcal{C}$ from Indri[1] using default parameters without removing stop-words. We index sequences of length $|\mathcal{C}| = n = 23{,}468{,}782{,}575$ consisting of $N = 25{,}205{,}179$ documents and $\sigma = 39{,}177{,}922$ unique word tokens. The raw collection uses ≈ 426 GiB, which is reduced to 71 GiB of 26-bit binary term identifiers after tokenization. All experiments were run on a server equipped with 148 GiB of RAM and two Intel Xeon E5640 processors each with a 12 MiB L3 cache.

To evaluate query performance, we traverse the suffix tree over $\mathcal{C}$ and assign phrases according to: a band $1 \leq b \leq 5$ where $10^b \leq N_{\mathcal{P}} < 2 \times 10^b$; a band $b \leq l \leq 7$, where the phrase's least frequent term occurs between 10^l and 2×10^l times in $\mathcal{C}$; and by phrase length $2 \leq m \leq 6$. Each bucket was capped at 1,000 phrases; because of the three dimensional nature of the categorization, some had fewer. All reported timings are median per-query elapsed times, with all index components fully memory-resident.

Index Sizes and Implementation. Figure 1 shows space use. The suffix array-based indexes are roughly the size of $\mathcal{C}$, or larger. The CSA shared by all suffix array methods requires 21 GiB. The SORT method adds an uncompressed *DA* array of 70 GiB, whereas the WT method uses a wavelet tree over *DA* (WTD), which is stored

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

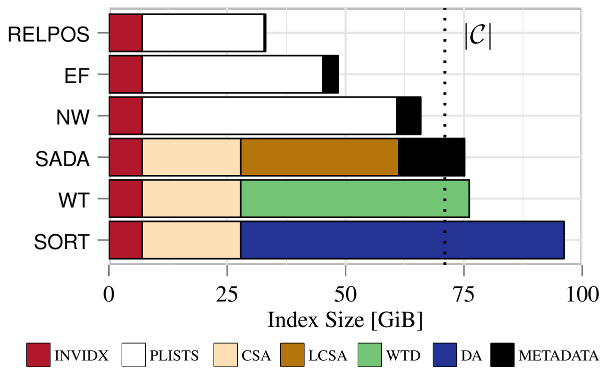

Figure 1: Space usage for indexes for GOV2. The dotted line shows the size of the tokenized collection.

using hybrid bitvectors [8], and requires only 49 GiB. The SADA method uses a local CSA (denoted LCSA in Figure 1) for each document; these are implemented using the method described by Gog et al. [6]. The metadata for SADA includes the two RMQ structures. These index arrangements reflect the current (practical) state-of-the-art for succinct indexes for the sorted document listing with frequencies problem. The components were partially provided by and implemented using the current version of the sdsl library [6]; our additional code is also available.[2]

The three inverted file-based indexes share the same document-level inverted index (INVIDX), implemented with uniform partitioned EF lists (UEF) with blocksize 128. The document identifiers and frequencies require 5.5 GiB, matching the values reported by Ottaviano and Venturini [12][3]. The inverted indexes additionally require 2 GiB metadata used for list offsets, document permutations, and WAND list max scores. The positional data is represented using absolute offsets, but for comparison we also include the size of an positional index that uses relative positions (RELPOS). This approach reduces the size significantly, but is slower than the absolute positional index [17]. The absolute index requires additional metadata to map absolute position offsets in $\mathcal{C}$ to document identifiers, implemented using a uncompressed bitvector of length n which marks each document boundary in $\mathcal{C}$. Constant time rank operations are used to achieve the mapping [11]. The nextword index (NW) stores absolute positions for all 473,366,430 bi-grams in GOV2. While there are techniques that only partially store lists [3, 13, 19], we measure the exhaustive case in which all bi-grams are indexed. The position lists of NW are stored using UEF codes, and require 55 GiB, still less than the three suffix-based indexes.

Document Level Retrieval. As a preliminary, we compare the performance of suffix array and inverted file-based methods in the context of bag-of-words conjunctive Boolean queries. Four indexes are used: the UEF and WT methods already described; plus regular Elias-Fano (EF) and OptPforDelta (OP4) encoded postings lists. The latter two require indexes of 8.3 GiB and 5.7 GiB respectively; and in the EF, UEF, and OP4 methods, intersection is achieved via the set-versus-set (SvS) approach. The WT index supports conjunctive queries by performing *intersection* operations, as described by Gagie et al. [5]. For each query term, the range $SA[sp, ep]$ is determined. These are then processed simultaneously using the wavelet tree over DA, to determine document identifiers which contain all terms. This approach has not yet been compared empirically to inverted file-based intersection approaches, hence our interest.

Queries	EF	UEF	OP4	WT
TREC 2005	0.92	1.51	1.28	77.78
TREC 2006	2.32	3.71	3.12	148.90

Table 1: Median conjunctive Boolean bag-of-words retrieval times, in milliseconds, over GOV2.

Band	EF	NW	SORT	WT	SADA
$b = 1$	0.09	0.08	0.11	0.37	0.37
$b = 2$	0.69	0.38	0.12	1.59	2.65
$b = 3$	6.43	1.20	0.22	9.09	25.36
$b = 4$	62.69	2.09	1.34	34.36	222.41
$b = 5$	522.20	52.66	13.61	226.17	1922.13

Table 2: Median phrase materialization times, in milliseconds, over GOV2, using 1,000 queries in each bucket except when $b = 5$ (421 queries), with $10^b \leq N_{\mathcal{P}} < 2 \times 10^b$ in the bth band. The pattern length is fixed at $m = 3$, and the smallest list size band at $l = b+2$ for queries in the bth band.

All queries of length $m \geq 2$ in the TREC 2005 and 2006 Terabyte Track efficiency tasks are used in this experiment, 34,495 and 94,253 queries respectively. Table 1 shows median query times, in milliseconds. The relative times of the inverted file-based indexes are broadly similar to those reported in recent studies [7, 12], except that our OP4 index is faster than the UEF index. This is in contrast to what was reported by Ottaviano and Venturini [12], a difference that we attribute to optimizations done in the NEQ skip method they used. All studies to date agree that EF is faster than OP4, but requires more space.

The three inverted file-based indexes outperform the WT index by a considerable margin. This is a consequence of the random memory accesses required to work the wavelet tree, compared to fast sequential processing of postings lists in the inverted file indexes. That is, the WT index is not competitive in either space or time. However, the WT index can also answer phrase queries, whereas additional positional information is required before inverted file indexing schemes can do the same (Figure 1).

Phrase Materialization. We turn to our main interest – the query-time generation of postings lists for phrases, ready for incorporation in ranked retrieval such as WAND-based evaluation of the BM25 scoring regime. Similarity computations using WAND can be performed efficiently, meaning that materializing additional postings lists must also be fast if it is not to dominate execution times. For example, Ottaviano and Venturini [12] report average BM25 computation times of ≈ 9 milliseconds for GOV2.

Table 2 shows median query times to materialize phrase lists for synthetic queries of length $m = 3$, categorized according to the band corresponding to the result size $N_{\mathcal{P}}$. For each band b, we fix the smallest position list size l to be $b + 2$. That is, the smallest postings list for each query is $\approx$ 100 times larger than $N_{\mathcal{P}}$ for that query. For small b, the cost of phrase materialization using all methods is within the cost of performing a WAND computation. As b becomes larger, the WT method becomes uncompetitive; once $b = 5$, both SADA and EF require too much time to be included in a similarity computation. The simple SORT method and NW index remain competitive. Surprisingly, the fastest index for bands $b \geq 2$ is the simple SORT index, which copies and sorts parts of DA. Unlike SADA and WT, the performance of SORT is dependent on $ep - sp$, which can be much larger than $N_{\mathcal{P}}$. But SORT does not perform random accesses, and instead makes use of fast localized

[2] `https://github.com/mpetri/pos-cmp`

[3] Yan et al. [21] report 4.1 GiB but the basis of this is unclear, and may involve stopping or other index reduction techniques.

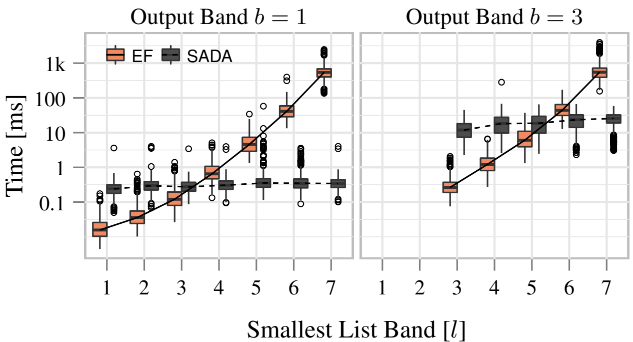

Figure 2: Phrase materialization times for EF and SADA, in milliseconds over GOV2, for queries where $10^b \leq N_{\mathcal{P}} < 2 \times 10^b$ for $b = 1$ and $b = 3$ for smallest list sizes $l \in b \ldots 7$. Patterns lengths are in the range $m \in 2 \ldots 6$.

integer sorting. Its drawback is the space required – 98 GiB, more than the other indexes.

In the previous experiment the size of the input for EF is fixed to be a constant ratio of the output size. The performance of list intersection-based methods depends on the size of the smallest list to be intersected, whereas all suffix array based methods only depend on the size of $N_{\mathcal{P}}$. Figure 2 shows the phrase materialization cost for patterns with $b = 1$ and $b = 3$, varying l, the minimum list length band, over $b \leq l \leq 7$. The time to process a phrase query using SADA depends primarily on the output size, rather than the lengths of the terms' position lists. In contrast, the EF run time significantly increases as l grows. In particular, EF is faster than SADA for small l (for example, for $b = 3$ and $l = 4$, EF takes 1.22 milliseconds, compared to 18.16 for SADA), but is more than an order of magnitude slower for phrases containing no infrequent terms.

Additional Trade-offs. We have evaluated a selection of index types in our phrase list comparison. Other combinations have also been proposed, with different time and space trade-offs. The nextword index (NW) in our experiments stores positional lists for all unique word bi-grams in $\mathcal{C}$. Williams et al. [19] examine hybrid schemes, in which only certain bi-gram lists are stored, reducing the space required. Similarly, Petri et al. [13] propose a mixed arrangement which pre-computes document/frequency lists up to a certain size threshold. Smaller lists are materialized at query time by processing *DA*, similar to the SORT index. A nextword index could also be used to materialize missing lists in this scheme. Another option is to avoid storing *DA* explicitly, and extract document numbers from the CSA by storing an additional bitvector marking document boundaries. This change would greatly reduce the storage cost of the index, but would add $ep - sp$ calls to suffix array values and rank operations, adversely affecting query time. That is, this method can only be viable for small $\langle sp, ep \rangle$ ranges.

Construction Cost. Our primary focus has been on phrase list materialization times. But construction costs are also a factor to be considered when choosing an index. The cost of building the suffix array-based indexes is an order of magnitude higher than inverted file-based indexing methods [6], and constructing *SA* for large parsings requires RAM not available in commodity hardware.

5. CONCLUSION

We compare inverted file indexes to suffix-based alternatives. The WT index is uncompetitive in both time and space for conjunctive Boolean retrieval; and all suffix-based indexes are larger than their inverted file-based counterparts. For phrase components, the SADA and WT methods are fast to materialize short lists, but slow considerably when there are many answers. Regular positional list intersection can produce phrase lists efficiently if the smallest intersected list is short. The nextword index (NW) is smaller than all suffix array indexes, and provides materialization times which are reasonably fast. The simple SORT index processes phrase queries rapidly even for large b, but uses nearly 100 GiB RAM. These various relativities are determined by the number of answers and the frequency of the smallest input term, and in future work we will more fully categorize the respective zones of applicability.

Acknowledgment. This work was funded by the Australian Research Council's *Discovery Project* scheme (project DP140103256), and by the Victorian Life Sciences Computation Initiative (grant VR0052), an initiative of the Victorian Government, Australia.

References

[1] D. Arroyuelo, S. González, M. Marín, M. Oyarzún, and T. Suel. To index or not to index: Time-space trade-offs in search engines with positional ranking functions. In *Proc. SIGIR*, pages 255–264, 2012.

[2] 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.

[3] A. Broschart and R. Schenkel. High-performance processing of text queries with tunable pruned term and term pair indexes. *ACM Trans. Inf. Sys.*, 30(1):5, 2012.

[4] J. S. Culpepper and A. Moffat. Efficient set intersection for inverted indexing. *ACM Trans. Inf. Sys.*, 29(1):1, 2010.

[5] T. Gagie, G. Navarro, and S. J. Puglisi. New algorithms on wavelet trees and applications to information retrieval. *Theor. Comp. Sc.*, 426-427:25–41, 2012.

[6] S. Gog, T. Beller, A. Moffat, and M. Petri. From theory to practice: Plug and play with succinct data structures. In *Proc. SEA*, pages 326–337, 2014.

[7] A. Kane and F. W. Tompa. Skewed partial bitvectors for list intersection. In *Proc. SIGIR*, pages 263–272, 2014.

[8] J. Kärkkäinen, D. Kempa, and S. J. Puglisi. Hybrid compression of bitvectors for the FM-index. In *Proc. DCC*, pages 302–311, 2014.

[9] D. Lemire and L. Boytsov. Decoding billions of integers per second through vectorization. *Soft. Prac. & Exp.*, 45(1):1–29, 2015.

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

[11] G. Navarro. Spaces, trees and colors: The algorithmic landscape of document retrieval on sequences. *ACM Comp. Surv.*, 46(4.52), 2014.

[12] G. Ottaviano and R. Venturini. Partitioned Elias-Fano indexes. In *Proc. SIGIR*, pages 273–282, 2014.

[13] M. Petri, A. Moffat, and J. S. Culpepper. Score-safe term-dependency processing with hybrid indexes. In *Proc. SIGIR*, pages 899–902, 2014.

[14] K. Sadakane. New text indexing functionalities of the compressed suffix arrays. *J. Alg.*, 48(2):294–313, 2003.

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

[16] T. Schnattinger, E. Ohlebusch, and S. Gog. Bidirectional search in a string with wavelet trees. In *Proc. CPM*, pages 40–50, 2010.

[17] D. Shan, W. X. Zhao, J. He, R. Yan, H. Yan, and X. Li. Efficient phrase querying with flat position index. In *Proc. CIKM*, pages 2001–2004, 2011.

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

[19] H. E. Williams, J. Zobel, and D. Bahle. Fast phrase querying with combined indexes. *ACM Trans. Inf. Sys.*, 22(4):573–594, 2004.

[20] H. Yan, S. Ding, and T. Suel. Compressing term positions in web indexes. In *Proc. SIGIR*, pages 147–154, 2009.

[21] H. Yan, S. Ding, and T. Suel. Inverted index compression and query processing with optimized document ordering. In *Proc. WWW*, pages 401–410, 2009.