Inverted Indexes for Phrases and Strings[*]

Manish Patil
Department of CS
Louisiana State University
USA
mpatil@csc.lsu.edu

Sharma V. Thankachan
Department of CS
Louisiana State University
USA
thanks@csc.lsu.edu

Rahul Shah
Department of CS
Louisiana State University
USA
rahul@csc.lsu.edu

Wing-Kai Hon
Department of CS
National Tsing Hua University
Taiwan
wkhon@cs.nthu.edu.tw

Jeffrey Scott Vitter
Department of EECS
The University of Kansas
USA
jsv@ku.edu

Sabrina Chandrasekaran
Department of CS
Louisiana State University
USA
schand7@lsu.edu

ABSTRACT

Inverted indexes are the most fundamental and widely used data structures in information retrieval. For each unique word occurring in a document collection, the inverted index stores a list of the documents in which this word occurs. Compression techniques are often applied to further reduce the space requirement of these lists. However, the index has a shortcoming, in that only predefined pattern queries can be supported efficiently. In terms of string documents where word boundaries are undefined, if we have to index all the substrings of a given document, then the storage quickly becomes quadratic in the data size. Also, if we want to apply the same type of indexes for querying phrases or sequence of words, then the inverted index will end up storing redundant information. In this paper, we show the first set of inverted indexes which work naturally for strings as well as phrase searching. The central idea is to exclude document d in the inverted list of a string P if every occurrence of P in d is subsumed by another string of which P is a prefix. With this we show that our space utilization is close to the optimal. Techniques from succinct data structures are deployed to achieve compression while allowing fast access in terms of frequency and document id based retrieval. Compression and speed tradeoffs are evaluated for different variants of the proposed index. For phrase searching, we show that our indexes compare favorably against a typical inverted index deploying position-wise intersections. We also show efficient top-k based retrieval under relevance metrics like frequency and *tf-idf*.

Categories and Subject Descriptors

H.3.1 [**Information Storage and Retrieval**]: Content Analysis and Indexing—*Indexing methods*; I.7.3 [**Document and Text Processing**]: Index Generation

General Terms

Algorithms, Experimentation

1. INTRODUCTION

The most popular data structure in the field of Information Retrieval is the inverted index. For a given collection of documents, the index is defined as follows. Each word in this collection is called a *term* and corresponding to each term we maintain a list, called *inverted list*, of all the documents in which this word appears. Along with each document in this list we may store some score which indicates how important the document is with respect to that word. Different variants of the inverted index sort the documents in the inverted lists in a different manner. For instance, the sorting order may be based on the document ids or the scores. Compression techniques are often applied to further reduce space requirement of these lists. However, inverted index has a drawback that it can support queries only on predefined words or terms. As a result, it cannot be used to index documents without well-defined word boundaries.

Different approaches have been proposed to support phrase searching using an inverted index. One strategy is to maintain the position information in the inverted list. That is, for each document d in the inverted list of a word w, we store the positions at which w occurs in d. The positions corresponding to each d in the list can be sorted so as to achieve compression (using encoding functions like gap, gamma, or delta) [15]. To search a phrase, we first search for all the words in the phrase and obtain the corresponding inverted lists. The positions of each word within a document are extracted, so that we can then apply an intersection algorithm to retrieve those documents where these words are appearing in the same order as in the phrase. Another (naive) approach is to store inverted lists for all possible phrases, however, the resulting index size will be very large thus prohibiting its use in practice [38]. Different heuristics are proposed in this respect, such as maintaining the inverted lists only for popular phrases, or maintaining inverted lists of all

[*]This work is supported in part by Taiwan NSC Grant 99-2221-E-007-123 (W. Hon) and US NSF Grant CCF–1017623 (R. Shah and J. S. Vitter).

SIGIR'11, July 24–28, 2011, Beijing, China.
Copyright 2011 ACM 978-1-4503-0757-4/11/07 ...$10.00.

phrases up to some fixed number (say h) of words. Another approach is called "next-word index" [36, 3, 4, 37], such that corresponding to each term w, a list of all the terms which occurs immediately after w is maintained. This approach will double the space, but it can support searching of any phrase with two words efficiently. Nevertheless, when the phrase goes beyond two words, we have to fall back to the intersection algorithm.

In this paper, we first introduce a variant of inverted index which naturally works for string as well as phrase searching. Our index does not assume any restrictions on the length or the popularity of the phrases. In addition, by avoiding the use of the intersection algorithm we achieve provable bounds for the query answering time with respect to the output size. Furthermore, we show different heuristics and compression techniques to make our index space-efficient. For a collection of English documents, the size of our index for strings and for phrases are ≈ 5 times and ≈ 2 times, respectively, of that of the input data, while it can support document retrievals in 10-40 microseconds per document in ranked order.

2. RELATED WORK

Suffix trees and suffix arrays are efficient data structures which can be used to index a text and support searching for any arbitrary pattern. These data structures can be maintained in linear space and can report all the occurrence of a pattern P in optimal (or nearly optimal) time. The space-efficient versions of suffix trees and suffix arrays are called compressed suffix trees and compressed suffix arrays, respectively, which take space close to the size of the indexed text. From a collection $\mathcal{D}$ of $|\mathcal{D}|$ documents $\{d_1, d_2, ..., d_{|\mathcal{D}|}\}$ of total length n, the problem of reporting documents containing a query pattern P is called the "document listing" problem. This problem was first studied by Matias et al. [21], where they proposed a linear space index with $O(p \log n + |output|)$ query time; here, p denotes the length of the input pattern P and $|output|$ denotes the number of the qualified documents in the output. An index with optimal $O(p + |output|)$ query time was later achieved in [24]. Sadakane [31] showed how to solve the document listing problem using succinct data structures, which take space very close to that of the compressed text. He also showed how to compute the *tf-idf* [2] of each document with the proposed data structures. Similar work was also done by Välimäki and Mäkinen [33] where they derived alternative succinct data structures for the problem.

In many practical situations, we may be interested in only a few documents which are highly relevant to the query. Relevance ranking refers to the ranking of the documents in some order, so that the result returned first is what the user is most interested. This can be the document where the given query pattern occurs most number of times (frequency). The relevance can also be defined by a similarity metric, such as the proximity of the query pattern to a certain word or to another pattern. This problem is modeled as top-k document retrieval, where the task is to retrieve the k highest scoring documents based on some score function. An $O(n \log n)$ words index has been proposed in [16] with $O(p + \log |\mathcal{D}| \log \log |\mathcal{D}| + k)$ query time. Hon et al. [18] proposed a linear-space index ($O(n)$ words) with nearly optimal $O(p + k \log k)$ query time. Yet, the constants hidden in the space bound restricts its use in practice. Culpepper et al. [8] proposed a space-efficient practical index based on wavelet trees [13], but their query algorithm is based on a heuristic, so that it does not guarantee any worst-case query performance.

The most popular ranking function in web search applications is *tf-idf* [2]. Under the *tf-idf* model, Persin et al. [28] give different heuristics to support top-k ranked retrieval when the inverted lists are sorted in decreasing order of the *tf* score. Various generalizations of this are studied by Anh and Moffat [1] under the name "impact ordering". In [25], Navarro and Puglisi showed that wavelet trees can be used for maintaining dual-sorted inverted lists corresponding to a word, where the documents can efficiently be retrieved in score order or in document id order. Recently, Hon et al. [17] proposed an index for answering top-k multi-pattern queries. On a related note, top-k color query problems (with applications in document retrieval) have been studied in [12, 20].

3. PRELIMINARIES

3.1 Suffix Trees and Compressed Suffix Trees

Given a text $T[1...n]$, a substring $T[i...n]$ with $1 \leq i \leq n$ is called a suffix of T. The lexicographic arrangement of all n suffixes of T in a compact trie is known as the *suffix tree* of T [35], where the ith leftmost leaf represents the ith lexicographically smallest suffix. Each edge in the suffix tree is labeled by a character string and for any node u, $path(u)$ is the string formed by concatenating the edge labels from root to u. For any leaf v, $path(v)$ is exactly the suffix corresponding to v. For a given pattern P, a node u is defined as the *locus node* of P if it is the node u closest to the root such that P is a prefix of $path(u)$; such a node can be determined in $O(p)$ time, where p denotes the length of P. The *generalized suffix tree* (GST) is a compact trie which stores all suffixes of all strings in a given collection $\mathcal{D}$ of strings. The drawback of the suffix tree is its huge space consumption, which requires $O(n \log n)$ bits in theory. Yet, it can perform pattern matching in optimal $O(p + |output|)$ time, where $|output|$ is the number of occurrences of P in T. Compressed suffix tree (CST) is a space-efficient version of suffix tree. Several variants of CSTs have been proposed to date [23, 14, 32, 30, 11, 27, 34, 6, 26].

3.2 Range Minimum/Maximum Query (RMQ)

Let $A[1...n]$ be an array of length n. The RMQ index is a linear-space data structure which can return the position and the value of the minimum (maximum) element in any subrange $A[i..j]$ such that $0 \leq i \leq j \leq n$. Although solving RMQ can be dated back from Chazelle's original paper on range searching [7], many simplifications [5] and improvements have been made, culminating in Fischer et al.'s $2n + o(n)$ bit data structure [9, 10]. All these schemes can answer RMQ in $O(1)$ time. We shall use RMQ data structures extensively to report the desired documents while answering our query. The basic result is captured in the following lemma [18]:

LEMMA 1. *Let A be an array of numbers. We can preprocess A in linear time and associate A with a linear-space RMQ data structure such that given a set of t non-overlapping ranges $[L_1, R_1], [L_2, R_2], ..., [L_t, R_t]$, we can find the largest (or the smallest) k numbers in $A[L_1, R_1] \cup A[L_2, R_2] \cup ... \cup A[L_t, R_t]$ in $O(t + k \log k)$ time.*

4. INVERTED INDEX FOR STRINGS AND PHRASES

In traditional inverted indexes, phrase queries are performed by first retrieving the inverted list for each word in the phrase and then applying an intersection algorithm to retrieve those documents in which the words in are appearing in the same order as in the phrase. Unfortunately, there is no efficient algorithm known which performs this intersection in time linear to the size of the output. Another limitation of the traditional inverted indexes is that they do not support string documents where there is no word demarcation (that is, when a query pattern can begin and end anywhere in the document). A naive approach to address these issues is to maintain inverted lists for all possible phrases (or strings). In the next subsection, we introduce a simple index that is based on a suffix tree and augments this with the inverted lists. This index can answer the queries in optimal time, however, the space is a factor of $|\mathcal{D}|$ away from the optimal. As phrase is a special case of a string (that is, string that starts and ends at word boundaries), we will explain our indexes in terms of strings.

4.1 Inverted Lists

Let $\mathcal{D}$=$\{d_1, d_2, ..., d_{|D|}\}$ be the collection of documents of total length n drawn from an alphabet set Σ, and Δ be the generalized suffix tree of $\mathcal{D}$. Let u be the locus node of a pattern P. Now a naive solution is to simply maintain an inverted list for the pattern corresponding to $path(u)$ for all internal nodes u in Δ. The list associated with a node u consists of pairs of the form $(d_j, score(path(u), d_j))$ for $j = 1, 2, 3, ..., |\mathcal{D}|$, where the score of a document d_j with respect to pattern $P = path(u)$ is given by $score(path(u), d_j)$. We assume that such a score is dependent only on the occurrences of P in the document d_j. An example of such a score metric is frequency, so that $score(P, d_j)$ represents the number of occurrences of pattern P in document d_j. For a given online pattern P, the top-k highest scoring documents can be answered by reporting the first k documents in the inverted list associated with the locus node of P, when the inverted lists are sorted by score order. Since the inverted list maintained at each node can be of length $|\mathcal{D}|$, the total size of this index is $O(n|\mathcal{D}|)$. Though this index offers optimal query time, it stores the inverted list for all possible strings. In the next subsection we show how the inverted lists can be stored efficiently in a total of $O(n)$ space.

4.2 Conditional Inverted Lists

The key idea which leads to $O(n)$ storage for inverted lists is the selection of nodes in the suffix tree for which inverted lists are actually maintained. We begin with the following definitions.

- Maximal String: A given string P is *maximal* for document d, if there is no other string Q such that P is a prefix of Q and every occurrence of P in d is subsumed by Q.
- Conditional Maximal String: Let Q be a maximal string for which P is a prefix and there is no maximal string R such that R is in between P and Q. That is P is a prefix of R and R is a prefix of Q. Then we call Q a *conditional maximal string* of P.

Consider the following sample documents d_1, d_2, and d_3:

- d_1: *This is a cat. This is not a monkey. This is not a donkey.*
- d_2: *This is a girl. This is a child. This is not a boy. This is a gift.*
- d_3: *This is a dog. This is a pet.*

Note that "*This is* " is maximal in d_1 as well as d_2, but not in d_3. The conditional maximal strings of "*This is* " in d_1 are "*This is a cat ... donkey.*" and "*This is not a* ". The conditional maximal strings of "*This is* " in d_2 are "*This is a* " and "*This is not ... gift.*".

Lemma 2. *The number of maximal strings in a document d_j is less than $2|d_j|$.*

Proof. Consider the suffix tree Δ_j of document d_j. Then for each maximal string P in d_j, there exists a unique node u in Δ_j such that $path(u) = P$. Thus the number of maximal strings in d_j is equal to the number of nodes in Δ_j. □

Lemma 3. *For a given pattern P, we have $score(P, d_j) = score(P_i, d_j)$, where P_i is the shortest maximal string in d_j with P as prefix. If such a string P_i does not exist, then $score(P, d_j) = 0$.*

Proof. As P_i is the shortest maximal string in d_j with P as prefix, every occurrence of a pattern P in d_j is subsumed by an occurrence of pattern P_i. Hence both patterns will have the same score with respect to document d_j, with $score(P, d_j) = 0$ signifying that the pattern P does not occur in d_j. □

Lemma 4. *For every maximal string Q($\neq$ empty string) in d_j, there exists a unique maximal string P such that Q is a conditional maximal string of P.*

Proof. Corresponding to each maximal string Q in d_j, there exists a node u in Δ_j (suffix tree of document d_j) such that $Q = path(u)$. The lemma follows by setting $P = path(parent(u))$, where $parent(u)$ denotes the parent of u in Δ_j. □

The number of maximal strings in $\mathcal{D}$=$\{d_1, d_2, ..., d_{|D|}\}$ is equal to the number of nodes in Δ (Lemma 2). In the context of maximal strings, the index in Section 4.1 maintains inverted lists for all maximal strings in $\mathcal{D}$. However, $score(P, d_j)$ depends only on pattern P and document d_j. This gives the intuition that, for a particular document d_j, instead of having entries in inverted lists corresponding to all maximal strings in $\mathcal{D}$, it is sufficient to include d_j in the inverted lists of only those strings which are maximal in d_j. Thus, for each document d_j, there will be at most $2|d_j|$ entries in all inverted lists, so that the total number of such entries corresponding to all documents is at most $\sum_{j=1}^{|\mathcal{D}|} 2|d_j| = O(n)$. However, the downside of this change is that the simple searching algorithm used in Section 4.1 can no longer serve the purpose. Therefore, we introduce a new data structure called "conditional inverted lists", which is the key contribution of this paper.

From now onwards, we refer to the maximal strings by the pre-order rank of the corresponding node in Δ. That is $P_i = path(u_i)$, where u_i is a node in Δ with pre-order rank i. In contrast to the traditional inverted list, the conditional inverted list maintains $score(P_i, d_j)$ only if P_i is maximal in

d_j. Moreover $score(P_i, d_j)$ is maintained not with P_i, but instead with P_x, such that P_i is a conditional maximal string of P_x in d_j. Therefore, u_x will be a node in the path from root to u_i. Formally, the conditional inverted list is an array of triplets of the form $(string\ id, document\ id, score)$ sorted in the order of string-ids, where the string-id is pre-order rank of a node in Δ. A key observation is the following: The conditional inverted list of a string P_x has an entry $(i, j, score(P_i, d_j))$ if and only if P_i is a conditional maximal string of P_x in document d_j. From the earlier example, the conditional inverted list of "*This is* " has entries corresponding to the following strings. We assign a *string id* to each of these strings (for simplicity) and let the *score* of a string corresponding to a document be its number of occurrences in that document.

"*This is a cat ... donkey.*" ($string\ id = i_1$, $score$ in $d_1 = 1$)
"*This is not a* " ($string\ id = i_2$, $score$ in $d_1 = 2$)
"*This is a* " ($string\ id = i_3$, $score$ in $d_2 = 3$)
"*This is not a ... gift.*" ($string\ id = i_4$, $score$ in $d_2 = 1$)

Since the *string ids* are based on the lexicographical order, $i_3 < i_1 < i_2 < i_4$. Then the conditional inverted list associated with the string "*This is* " is given below. Note that there is no entry for d_3, since "*This is* " is not maximal in d_3.

string id	i_3	i_1	i_2	i_4
document id	d_2	d_1	d_1	d_2
score	3	1	2	1

We also maintain an RMQ (range maximum query) structure over the *score* field in the conditional inverted lists so as to efficiently retrieve documents with highest score as explained later in following subsection.

Lemma 5. *The total size of conditional inverted lists is* $O(n)$.

Proof. Corresponding to each maximal string in d_j, there exists an entry in the conditional inverted list with document id j. Hence the number of entries with document id as j is at most $2|d_j|$ and the total size of conditional inverted lists is $O(\sum_{j=1}^{|\mathcal{D}|} 2|d_j|) = O(n)$. □

Lemma 6. *For any given node u in Δ and any given document d_j associated with some leaf in the subtree of u, there will be exactly one string P_i such that (1) P_i is maximal in d_j, (2) $path(u)$ is a prefix of P_i, and (3) the triplet $(i, j, score(P_i, d_j))$ is stored in the conditional inverted list of a node $u_x \neq u$, where u_x is some ancestor of u.*

Proof. Since there exists at least one occurrence of d_j in the subtree of u, Statements (1), (2), and (3) can be easily verified from the definition of conditional inverted lists. The uniqueness of P_i can be proven by contradiction. Suppose that there are two strings $P_i^{'}$ and $P_i^{''}$ satisfying all of the above conditions. Then $path(u)$ will be a prefix of $P_i^* = lcp(P_i^{'}, P_i^{''})$, where lcp is the longest common prefix. Then from the one-to-one correspondence that exists between maximal strings and nodes in suffix tree (Lemma 2), it can be observed that the lcp between two maximal strings in a document d_j is also maximal. Thus P_i^* is maximal in d_j and this contradicts the fact that, when $P_i^{'}$ (or $P_i^{''}$) is a conditional maximal string of P_x, there cannot be a maximal string P_i^*, such that P_i^* is a prefix of $P_i^{'}$ and P_x is a prefix of P_i^*. □

4.3 Answering Top-k Queries

Let P be the given online pattern of length p. To answer a top-k query, we first match P in Δ in $O(p)$ time and find the locus node u_i. Let $\ell = i$ and r be the pre-order rank of the rightmost leaf in the subtree of u_i. That is, P_ℓ and P_r represents the lexicographically smallest and largest maximal strings in $\mathcal{D}$ with $path(u_i)$ as a prefix. Then, all maximal strings with P as prefix can be represented by P_z, $\ell \leq z \leq r$. From Lemmas 4 and 6, for each document d_j which has an occurrence in the subtree of u_i, there exists a unique triplet with score $score(P, d_j)$ in the conditional inverted list of some ancestor node u_x of u_i with $string\ id \in [\ell, r]$. Now the top-k documents can be retrieved by first identifying such triplets and then retrieving the k highest scored documents.

Note that the triplets in the conditional inverted lists are sorted according to the string-ids. Hence by performing a binary search of ℓ and r in the conditional inverted list associated with each ancestor of u_i, we obtain t non-overlapping intervals $[\ell_1, r_1], [\ell_2, r_2], ..., [\ell_t, r_t]$, where $t < p$ is the number of ancestors of u_i. Using an RMQ (range maximum query) structure over the *score* field in the conditional inverted lists, the k triplets (thereby documents) corresponding to the k highest scoring documents can be retrieved in $O(t + k \log k)$ time (Lemma 1). Hence the total query time is $O(p) + O(t \log n) + O(t + k \log k) = O(p \log n + k \log k)$.

Theorem 1. *The String Inverted Index for a collection of documents $\mathcal{D} = \{d_1, d_2, ..., d_{|D|}\}$ of total length n can be maintained in $O(n)$ space, such that for a given pattern P of length p, the top-k document queries can be answered in $O(p \log n + k \log k)$ time.*

Note that the same structure can be used for document listing problem [24], where we need to list all the documents which has an occurrence of P. This can be answered by retrieving all the documents corresponding to the intervals $[\ell_1, r_1] \cup [\ell_2, r_2] \cup ... \cup [\ell_t, r_t]$ in the conditional inverted lists. Hence the query time is $O(p \log n + docc)$, where $docc$ is the number of documents containing P. If our task is to just find the number of such documents (counting, not listing), we may use $docc = \sum_{i=1}^{t}(r_i - \ell_i)$, and can answer the query in $O(p \log n)$ time.

Theorem 2. *Given a query pattern P of length p, the document listing queries for a collection of documents $\mathcal{D} = \{d_1, d_2, ..., d_{|D|}\}$ of total length n can be answered in $O(p \log n + docc)$ time, where docc is the number of documents containing P. The computation of docc (document counting) takes only $O(p \log n)$ time.*

The index described in this section so far is a generalized index for string documents. When word boundaries are well-defined and query patterns will be aligned with word boundaries as well, we can build the inverted index for phrases by replacing the generalized suffix tree with a word suffix tree. A word suffix tree is a trie of all suffixes which start from a word boundary. Now we maintain the conditional inverted lists corresponding to only those strings which start from a word boundary, thus resulting in huge space savings. We call this a phrase inverted index. Theorems 1 and 2 can be rewritten for phrase inverted index as follows.

THEOREM 3. *The Phrase Inverted Index for a collection of documents $\mathcal{D} = \{d_1, d_2, ..., d_{|D|}\}$ with total N suffixes, which start from a word boundary, can be maintained in $O(N)$ space, such that for a given pattern P of length p, the top-k, document listing, and document counting queries can be answered in $O(p \log N + k \log k)$, $O(p \log N + docc)$ and $O(p \log N)$ time, respectively.*

5. PRACTICAL FRAMEWORKS

In Section 4, we introduced the theoretical framework for our index. However, when dealing with the practical performance, the space and time analysis has to be more precise than merely a big-O notation. Consider a collection of English text documents of total length n, where each character can be represented in 8 bits (256 characters, including numbers and symbols corresponding to all ASCII values). Then the text can be maintained in $8n$ bits. The conditional inverted list can consist of at most $2n$ triplets and if each entry in the triplet is 32 bits (word in computer memory), then the total size of the conditional inverted lists can be as big as $(2n \times 3 \times 32)$ bits $= (24 \times 8n)$ bits $= 24 \times (datasize)$. Moreover, we also need to maintain the generalized suffix tree, which takes ≈20-30 times of the text size. Hence the total index size will be $\approx 50 \times (datasize)$. This indicates that the hidden constants in big-O notation can restrict the use of an index in practice especially while dealing with massive data.

In this section, we introduce a practical framework of our index when frequency is used as score metric. That is, $score(P, d_j)$ represents the number of occurrences of pattern P in document d_j. However, the ideas used can also be applied for other measures. Based on different tools and techniques from succinct data structures, we design three practical versions of our index (index-A, index-B, index-C) each successively improving the space requirements. We try to achieve the index compression by not sacrificing too much on the query times. Index-C takes only $\approx 5 \times (datasize)$, and even though it does not guarantee any theoretical bounds on query time, it outperforms the existing indexes [8] for top-k retrieval.

5.1 Index-A

Index-A is a direct implementation of our theoretical index from Section 4 with one change. As suffix tree is being used as an independent component in the proposed index, we replace it by compressed suffix tree (CST) without affecting the index operations and avoid the huge space required for suffix tree. We treat index-A as our base index as it does not modify the conditional inverted lists which form the core of the index.

5.2 Index-B

In this version, we apply different empirical techniques to compress each component of the triplets from the conditional inverted list separately.

- Compressing Document Array: Taking into account that fact that the total number of documents is $|D|$, we use only $\lceil \log |D| \rceil$ bits (instead of an entire word) per entry for the document value.

- Compressing Score Array: When pattern frequency is used as the score metric, score array consists of numbers ranging from 1 to n. The most space-efficient way to store this array would be to use exactly the minimal number of bits for each number with some extra information to mark the boundaries. But this approach may not be friendly in terms of retrieving the values. Our statistical studies showed that more than 90% of entries have frequency values less than 16 (which needs only 4 bits). This leads us to the heuristic for distributing frequency values into four categories: *a*) 1-4 bits, *b*) 5-8 bits, *c*) 9-16 bits, and *d*) 17-32 bits based on the actual number of bits required to represent each value. We use a simple wavelet tree structure [13] which first splits the array into two arrays, one with 1-8 bits and another with 9-32 bits, required per entry. Both arrays are further divided to cover the categories a, b and c, d, respectively. Each of the child nodes can be further divided into two. The values stored at the leaf nodes of the wavelet tree take only as many bits as represented by the category it belongs to. Further, we use rank-select [22, 29] structures on the bit vectors in the wavelet tree for fast retrieval of values.

- Compressing String-id Array: Since the entries in the conditional inverted lists are sorted based on string-id values, we observe that there will be many consecutive entries of the same string-id, each with different document-id. Therefore run-length encoding is a promising technique for string-id compression. In order to support fast retrieval of a particular string-id value, we again maintain additional bit vectors to keep track of which string-id values are stored explicitly and which values are eliminated due to repetition in the conditional inverted lists.

5.3 Index-C

In our final efforts to further reduce the space required for the index, the following two observations play an important role.

- Approximately 50% of the entries from all the conditional inverted lists in the index, have string-id corresponding to leaf node in Δ and have low score value (frequency of one).

- The document array, which is a part of the triplet in the conditional inverted lists, does not contribute in the process of retrieving top-k answers and is used only during reporting to identify the documents with highest score.

It follows from the first observation that pruning the conditional inverted list entries corresponding to leaf nodes would significantly reduce the index space. In particular, we do not store those triplets whose string-id field corresponds to a leaf node in Δ. The downside of this approach is that, the modified index will no longer be able to report the documents with frequency of one. However, this shortcoming can be justified by reductions in space, and algorithmic approach can be employed to retrieve such documents if needed.

From the second observation, we can choose to get rid of the document-id field and incur additional overhead during query time. Briefly speaking, the document-id in the triplet corresponding to an internal node (string-id = pre-order rank of that internal node) is not stored explicitly in the conditional inverted lists. The string-id of a triplet in a conditional inverted list associated with a node u_i is replaced by a pointer which points to another triplet associated with the *highest-descendent* node in the subtree of u_i with the *same* document-id. Now the triplets in the conditional inverted lists are sorted according to the value of this pointers. Retrieval of the document-id can be done in an online fashion by chasing pointers from an internal node up to the leaf corresponding to that document. (Details are deferred to the full paper).

Index-C makes use of both ideas simultaneously. Even though the modifications do not guarantee any theoretical bounds on query time (which can be $O(n)$ in worst case), we observed that index-C performs well in practice.

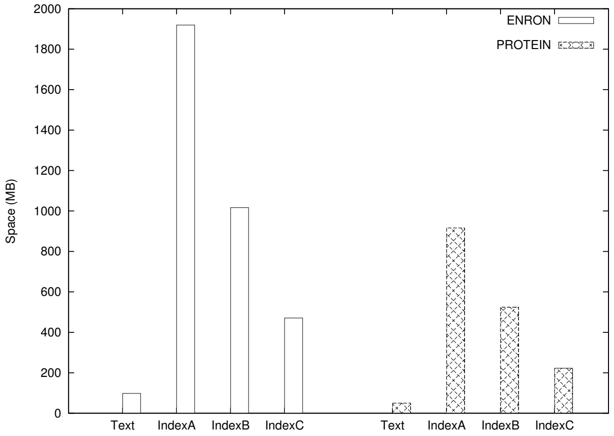

Figure 1: Space comparison of the indexes

6. EXPERIMENTAL ANALYSIS

We evaluated our new index and its compressed variants for space and query time using english texts and protein collections. ENRON is a ≈100MB collection of 48619 email messages drawn from a dataset prepared by the CALO Project (http://www.cs.cmu.edu/~enron/). PROTEIN is a concatenation of 141264 Human and Mouse protein sequences totaling ≈60MB (http://www.ebi.ac.uk/swissprot). We implemented all of the above indexes using the programming language C++, compiled with the g++ compiler version 4.2. Public code libraries at http://www.uni-ulm.de/in/theo/research/sdsl.html and http://pizzachili.dcc.uchile.cl/indexes.html are used to develop some of the components in the indexes. Our experiments were run on an Intel Core 2 Duo 2.26GHz machine with a 4GB RAM. The OS was MAC OS X using version 10.6.5. In the following, we first analyze the space-time tradeoffs for various indexes described in this paper. Then we empirically compare these indexes with the inverted index when word boundaries are well defined and query patterns are aligned on word boundaries.

6.1 Space-Time Tradeoffs

Figure 1 shows the space requirements for the original index and its compressed variants against input text size for both datasets. Reduction in the space requirements for index-B and index-C can be analyzed separately for the three key components of the indexes: Document array, score array and string-id-array. Figure 2 shows the space utilization of these components for each of the proposed indexes. For both document array and score array, even though it is possible to use the theoretically-minimal number of bits required per entry, it would result in a slowdown in the query time due to the lacking of efficient mechanisms for the retrieval of the array values. In index-B, recall that we try to keep the encoding simple and do not compress the data to the fullest extent so as to achieve reasonable compression and restrict query time within acceptable limit simultaneously. In particular, as most of the values in the score (frequency) array (≈ 97% for ENRON, ≈ 98% for PROTEIN) are less than 16, the proposed heuristic for compressing the score array in index-B achieves a very good practical performance. Out of three components, string-id array is the least compressible as its values correspond to the pre-order ranks of nodes in the suffix tree with ranges from 0 to $|T| = n$. We can utilize the fact that string-id array entries for a node are sorted in the increasing order by using difference encoding (such as gap) for efficient compression. However, such a method would naturally incur a query time overhead. Instead, as mentioned in the previous section, index-B makes use of the run-length encoding to represent the consecutive entries with the same string-id value, and was able to eliminate ≈ 30% string-id array entries for ENRON and ≈ 25% string-id array entries for PROTEIN in our experiments. Using these compression techniques, index-B is ≈ 10 times the text as compared to index-A (≈ 20 times text).

Recall that index-C does not store the document id for each entry explicitly to achieve space savings, at the expense of a slightly longer time to report the documents. Space savings are also achieved when we prune the inverted list entries corresponding to the leaf nodes, which account for 50% in ENRON and 55% in PROTEIN of the total number of entries. As a result, index-C improves further on index-B and takes only ≈ 5 times of the text in the space requirement.

For these experiments, 250 queries from ENRON and 125 queries from PROTEIN, which appear in at least 10 documents with frequency 2 or more, are generated randomly for pattern lengths varying from 3 to 10. This therefore forms a total of 2000 and 1000 sample queries for ENRON and PROTEIN, respectively. In addition, we ensure that the selected patterns of length 3 appear in at least 80 documents to observe the practical time in reporting top-k (k = 10, 20, ..., 80) documents. Figure 3 shows the average time required to retrieve $k = 10$ documents with highest score (frequency) for patterns with varying lengths. Average time required for retrieving documents in descending order of score (frequency) for a set of patterns with length 3 is shown in Figure 4 for varying k. These figures show that space savings achieved by the successive variants of our index (with increasing level of compression) will not hurt the query time to a great extent. A nearly linear dependance of query time on pattern length and k can also be observed from these figures. Matching the pattern P in compressed suffix tree Δ and binary search to obtain intervals in conditional inverted list of nodes in compressed suffix tree during top-k retrieval dominates the query time for index-A. Occasional slight drop in the query time for the indexes for increasing pattern length can be attributed to the binary search as it depends on the number of documents in which the query pattern is present. Query

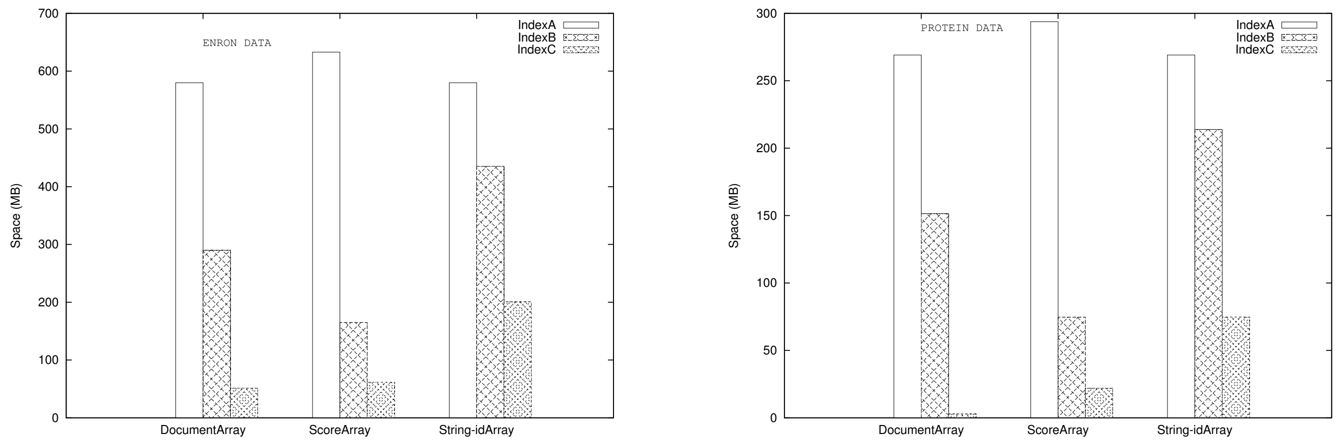

Figure 2: Compression achieved for each of three components in Conditional Inverted Lists

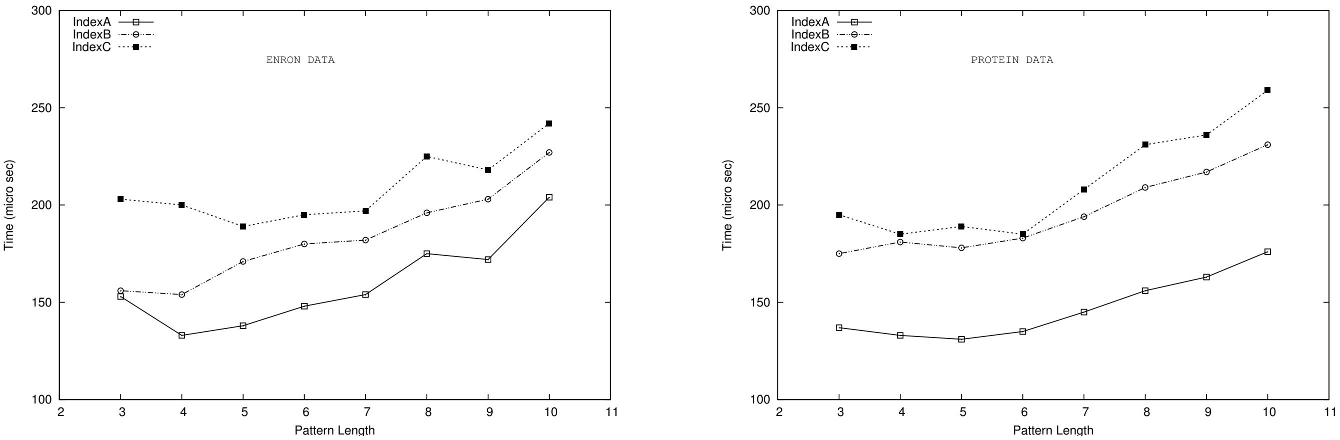

Figure 3: Mean time to report top-10 documents with highest frequency for a set of queries of varying lengths

timings for index-B closely follow to that of index-A, with decoding the score (frequency) values for possible top-k candidates being primarily responsible for the difference. Index-C has an additional overhead of decoding the document-id for each top-k answer to be reported. As a result, the gap in the query time of index-C with the other indexes should gradually increase with k, as is observed in the Figure 4.

6.2 Word/Term Based Search

In this subsection, we compare our phrase indexes with the traditional inverted index, highlighting the advantages of the former ones over the latter one. For a fair comparison, our proposed indexes in this subsection are built on the word suffix tree instead of the generalized suffix tree (Theorem 3) so as to support searching of only those patterns that are aligned with the word boundaries. We begin by comparing the query times. Traditional inverted index are known to be efficient for single-word searching. When the inverted lists are each sorted in descending order of score, ranked retrieval of documents would simply return the initial entries from the list corresponding to the query word. However, for efficient phrase searching, sorting the document lists by document-id (instead of score) would allow faster intersections of multiple lists. Figure 5 shows the time required for retrieving top-10 documents with highest score (frequency) for a set of phrases consisting of two and three words, respectively. Here, we generated 800 additional queries aligned on english word boundaries from ENRON. Traditional inverted index has its inverted lists sorted according to the document ids as mentioned, and we apply finger binary search [19] for intersecting multiple lists. We do not report the results

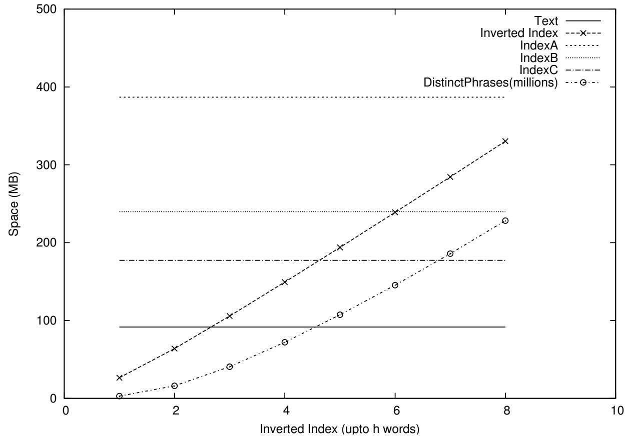

Figure 6: Space for the inverted index up to h words

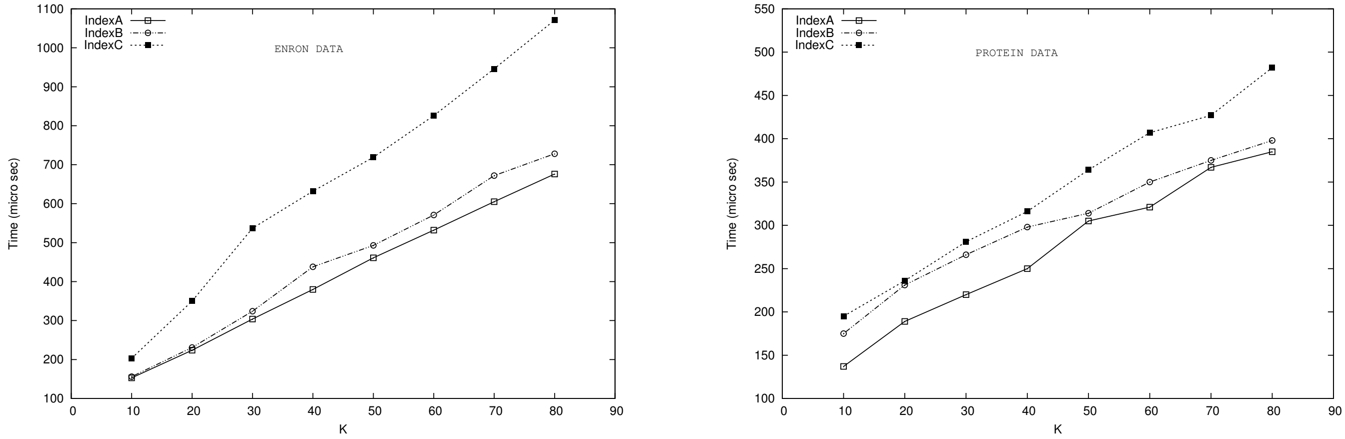

Figure 4: Mean time to report top-k documents with highest frequency for a set of queries with $|P| = 3$

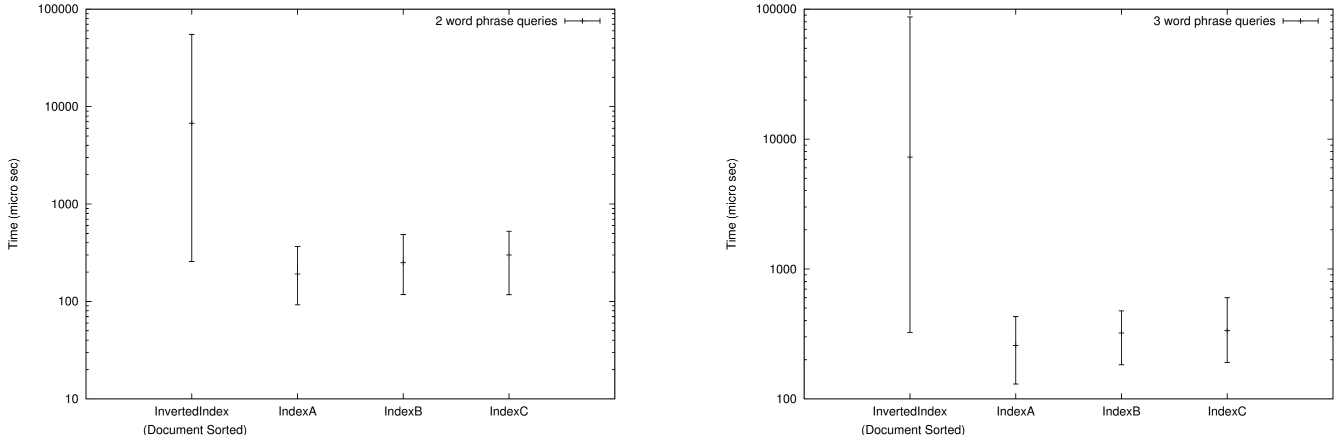

Figure 5: Time (High, Low, Mean) to report top-10 documents with highest frequency for a set of phrase queries with 2 words and 3 words

when inverted lists are sorted by score as the timings were significantly worse. Figure 5 show that our phrase indexes perform much better than the intersection-based retrieval, and the performance degradation in traditional inverted index would become more serious with the increase in words in a phrase query. Query times of our string/phrase indexes show that its query time for reporting top-10 documents is in the range of 100-400 microseconds thus achieving good practical performance.

A key point behind the widespread usage of the inverted index is that it can be stored in little space when compared with the size of input document collection; 20%-60% or more depending on whether it includes the position lists. One way to avoid the intersection of position lists in the phrase queries would be to store inverted list of all phrases up to some fixed number (say h) of words. Such an index still has to reply on intersection for phrases with more than h words. Figure 6 shows the space requirement for this variant of inverted index without the position lists. From the figure, it is clear that the space required for such a solution gradually increases with h and directly depends on the number of distinct phrases in the input text. In contrast, our phrase index supports phrase searching with arbitrary number of words. In the most space-efficient version of our phrase index (index-C), it takes just under 2 times of the input text in space. With gradual increase in space required, the traditional inverted index for phrases up to h words occupies more space than index-C for all $h \geq 5$. It is important to note that the traditional inverted index is maintained as an additional data structure along with the original text, whereas our proposed indexes are self indexes and do not need original text. Thus our phrase index compares favorably against the traditional inverted index for phrase searching in practice.

7. TOP-k TF-IDF QUERIES

In web search engines, *tf-idf* (term frequency–inverse document frequency) [2] is one of the most popular metric for relevance ranking. The query consists of multiple keywords (patterns) say $P_1, P_2, ..., P_m$ and the score of a document d, $score(d)$, is given by

$$score(d) = \sum_{i=1}^{m} tf(P_i, d) \times idf(P_i),$$

where *a*) $tf(P_i, d)$ denotes the number of occurrences of P_i in d, and *b*) $idf(P_i) = \log \frac{|\mathcal{D}|}{1+docc(P_i)}$, with $|\mathcal{D}|$ representing the total number of documents and $docc(P_i)$ representing the number of documents containing pattern P_i. Many other versions of this metric are available in the literature. For top-k document retrieval that is based on the *tf-idf* metric (with multiple query patterns), most of the existing solutions are based on heuristics. When the query consists of a single pattern, the inverted index with document lists sorted in score order can retrieve top-k documents in optimal time. However, for an m-pattern query (a query consisting of m patterns say $P_1, P_2, ..., P_m$), we may need the inverted lists sorted according to the document id as well. In this section, we introduce an exact algorithm and compare the results obtained by applying it to inverted index as well as our index (index-B). Although our algorithm does not guarantee any worst-case query bounds, the focus is to explore the capabilities of our index as a generalized inverted index. Along with our index, we make use of a wavelet tree [13] over the document array for its advantages in offering dual-sorting functionalities. We restrict the query patterns to be words in order to give a fair comparison between our index and the inverted index.

Suppose that N denotes the number of suffixes in the word suffix tree. Let $DA[1...N]$ be an array of document ids, such that $DA[i]$ is the document id corresponding to ith smallest suffix (lexicographically) in the word suffix tree. Note that each entry in DA takes at most $\lceil \log |D| \rceil$ bits to store. Therefore a wavelet tree *W-Tree* of DA can be maintained in $N \log |\mathcal{D}|(1 + o(1))$ bits. Now, given the suffix range $[\ell, r]$ of any pattern P, the term frequency $tf(P, d_j)$ for the document with id j can be computed by counting the number of entries in DA with $DA[i] = j$ and $\ell \leq i \leq r$. This query can be answered in $O(\log |\mathcal{D}|)$ time by exploring the orthogonal range searching functionality of *W-Tree*. Since term frequency in any document can be computed using *W-Tree*, we do not store the score (term frequency) array in index-B. This slightly compensates for the additional space overhead due to *W-Tree*. Inverse document frequency *idf* can be computed using Theorem 3. For simplicity, we describe the algorithm for two pattern queries (P_1 and P_2) as follows, and the algorithm can be easily extended for the general m-pattern queries. Let S_{ans} and S_{doc} be two sets of documents which are set to empty initially, and let d_1^k and d_2^k represents the kth highest scoring document corresponding P_1 and P_2, with *term frequency* as the score function and $score(d) = tf(P_1, d)\ idf(P_1) + tf(P_2, d)\ idf(P_2)$.

```
$S_{ans} = S_{doc} = \{\}, x = y = 1$
while $|S_{ans}| < k$ do
    if $score(d_1^x) \geq score(d_2^y)$ then
        $S_{doc} \leftarrow S_{doc} \cup d_1^x$ and $x \leftarrow x + 1$
    else
        $S_{doc} \leftarrow S_{doc} \cup d_2^y$ and $y \leftarrow y + 1$
    end if
    if $|S_{doc}| = 1, 2, 4, 8, 16, ...$ then
        $score_{max} = tf(P_1, d_1^x)\ idf(P_1) + tf(P_2, d_2^y)\ idf(P_2)$
        for each $d \in S_{doc}$ do
            if $score(d) \geq score_{max}$ and $d \notin S_{ans}$ then
                $S_{ans} \leftarrow S_{ans} \cup d$
            end if
        end for
    end if
end while
Choose $k$ documents in $S_{ans}$ with the highest $score$ value
```

The main idea of the algorithm is to maintain a list of candidate top-k documents in the set S_{doc}, and refine the candidate set by moving documents to the set S_{ans} from time to time. Each document in S_{ans} will have score higher than an imaginary $score_{max}$, and the set S_{ans} will always contain the highest scoring documents we have examined so far. The algorithm stops as soon as S_{ans} contains k documents, in which we report the top-k documents from the set.

Experimental Analysis

We compare the performance of our index against the traditional inverted index for answering 2-pattern queries using the algorithm described above. In the traditional inverted index, document lists are sorted either by score (frequency) or document-id. To apply the above heuristic, we need dual-sorted documents lists, where each list is sorted on both score as well as document-id. Score sorted lists support ranked retrieval of documents for individual patterns but *tf-idf* score can not be computed efficiently. If lists are sorted by document-id, though *tf-idf* score computation is faster, document retrieval in ranked order is not efficient. As a result we first duplicate the document lists for each of the pattern P_i and sort them as required. Figure 7 shows the mean time required for retrieving top-k documents for a set of 50 2-pattern queries for `ENRON` such that each pattern is highly frequent. As observed from the figure, query time for our index increases faster than that of the inverted index.

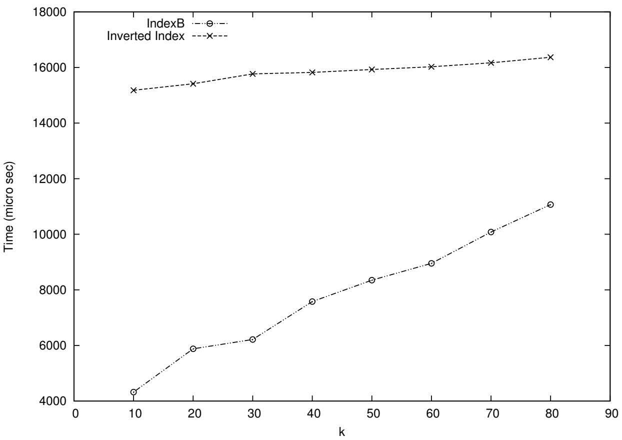

Figure 7: Mean time to report top-k documents for a set of 2-pattern queries

We remark that the major part of the query time used by the inverted index is on re-sorting the the document lists in which the query patterns occur. Thus, if the patterns are not too frequently occurring, the time spent on re-sorting is reduced, and the advantages of our index over the inverted index will vanish. Finally, the size of our index is ≈ 3.1 times of the text size.

8. CONCLUDING REMARKS

This paper introduces the first practical version of inverted index for string documents. The idea is to store lists for a selected collection of substrings (or phrases) in a conditionally sorted manner. Succinct data structures are used to represent these lists so as to reap benefits of dual sorting and achieve good top-k retrieval performance. We show

how top-k *tf-idf* based queries can be executed efficiently. Furthermore, our indexes show a space-time advantage over all of the traditional techniques for searching long phrases. While this is the first prototype, more research will certainly help in deriving structures with great practical impact.

9. REFERENCES

[1] V. Anh and A. Moffat. Pruned Query Evaluation using Pre-computed Impacts. In *ACM SIGIR*, pages 372–379, 2006.

[2] R. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval*. Addison Wesley, 1999.

[3] D. Bahle, H. E. Williams, and J. Zobel. Compaction Techniques for Nextword Indexes. In *SPIRE*, pages 33–45, 2001.

[4] D. Bahle, H. E. Williams, and J. Zobel. Optimised Phrase Querying and Browsing of Large Text Databases. In *ACSC*, pages 11–19, 2001.

[5] M. A. Bender and M. Farach-Colton. The Level Ancestor Problem Simplified. *Theoretical Computer Science*, 321(1):5–12, 2004.

[6] R. Cánovas and G. Navarro. Practical Compressed Suffix Trees. In *SEA*, pages 94–105, 2010.

[7] B. Chazelle. A Functional Approach to Data Structures and Its Use in Multidimensional Searching. *SIAM Journal on Computing*, 17(3):427–462, 1988.

[8] J. S. Culpepper, G. Navarro, S. J. Puglisi, and A. Turpin. Top-k Ranked Document Search in General Text Databases. In *ESA*, pages 194–205, 2010.

[9] J. Fischer and V. Heun. A New Succinct Representation of RMQ-Information and Improvements in the Enhanced Suffix Array. In *ESCAPE*, pages 459–470, 2007.

[10] J. Fischer, V. Heun, and H. M. Stühler. Practical Entropy-Bounded Schemes for $O(1)$-Range Minimum Queries. In *IEEE DCC*, pages 272–281, 2008.

[11] J. Fischer, V. Mäkinen, and G. Navarro. Faster Entropy-Bounded Compressed Suffix Trees. In *Theoretical Computer Science*, pages 5354–5364, 2009.

[12] T. Gagie, G. Navarro, and S. J. Puglisi. Colored Range Queries and Document Retrieval. In *SPIRE*, pages 67–81, 2010.

[13] R. Grossi, A. Gupta, and J. S. Vitter. High-Order Entropy-Compressed Text Indexes. In *ACM-SIAM SODA*, pages 841–850, 2003.

[14] R. Grossi and J. S. Vitter. Compressed Suffix Arrays and Suffix Trees with Applications to Text Indexing and String Matching. *SIAM Journal on Computing*, 35(2):378–407, 2005.

[15] A. Gupta, W. K. Hon, R. Shah, and J. S. Vitter. Compressed Data Structures: Dictionaries and Data-Aware Measures. *Theoretical Computer Science*, 387(3):313–331, 2007.

[16] W. K. Hon, M. Patil, R. Shah, and S. B. Wu. Efficient Index for Retrieving Top-k Most Frequent Documents. *Journal on Discrete Algorithms*, 8(4):402–417, 2010.

[17] W. K. Hon, R. Shah, S. V. Thankachan, and J. S. Vitter. String Retrieval for Multi-pattern Queries. In *SPIRE*, pages 55–66, 2010.

[18] W. K. Hon, R. Shah, and J. S. Vitter. Space-Efficient Framework for Top-k String Retrieval Problems. In *IEEE FOCS*, pages 713–722, 2009.

[19] F. K. Hwang and S. Lin. A Simple Algorithm for Merging Two Disjoint Linearly Ordered Sets. *SIAM Journal of Computing*, 1(1):31–39, 1972.

[20] M. Karpinski and Y. Nekrich. Top-K Color Queries for Document Retrieval. In *ACM-SIAM SODA*, pages 401–411, 2011.

[21] Y. Matias, S. Muthukrishnan, S. C. Sahinalp, and J. Ziv. Augmenting Suffix Trees, with Applications. In *ESA*, pages 67–78, 1998.

[22] J. I. Munro. Tables. In *FSTTCS*, pages 37–42, 1996.

[23] J. I. Munro, V. Raman, and S. S. Rao. Space Efficient Suffix Trees. In *FSTTCS*, pages 186–196, 1998.

[24] S. Muthukrishnan. Efficient Algorithms for Document Retrieval Problems. In *ACM-SIAM SODA*, pages 657–666, 2002.

[25] G. Navarro and S. J. Puglisi. Dual-Sorted Inverted Lists. In *SPIRE*, pages 309–321, 2010.

[26] E. Ohlebusch, J. Fischer, and S. Gog. CST++. In *SPIRE*, pages 322–333, 2010.

[27] E. Ohlebusch and S. Gog. A Compressed Enhanced Suffix Array Supporting Fast String Matching. In *SPIRE*, pages 51–62, 2009.

[28] M. Persin, J. Zobel, and R. S. Davis. Filtered Document Retrieval with Frequency-Sorted Indexes. *Journal of the American Society for Information Science*, 47(10):749–764, 1996.

[29] R. Raman, V. Raman, and S. S. Rao. Succinct Indexable Dictionaries with Applications to Encoding k-ary Trees and Multisets. In *ACM-SIAM SODA*, pages 233–242, 2002.

[30] L. Russo, G. Navarro, and A. Oliveira. Fully-Compressed Suffix Trees. In *LATIN*, pages 362–373, 2008.

[31] K. Sadakane. Space-Efficient Data Structures for Flexible Text Retrieval Systems. In *ISAAC*, pages 14–24, 2002.

[32] K. Sadakane. Compressed Suffix Trees with Full Functionality. *Theory of Computing Systems*, pages 589–607, 2007.

[33] N. Välimäki and V. Mäkinen. Space-Efficient Algorithms for Document Retrieval. In *CPM*, pages 205–215, 2007.

[34] N. Välimäki, V. Mäkinen, W. Gerlach, and K. Dixit. Engineering a Compressed Suffix Tree Implementation. *ACM Journal of Experimental Algorithmics*, 14, 2009.

[35] P. Weiner. Linear Pattern Matching Algorithms. In *SWAT*, pages 1–11, 1973.

[36] H. E. Williams, J. Zobel, and P. Anderson. What's Next? Index Structures for Efficient Phrase Querying. In *ADC*, pages 141–152, 1999.

[37] H. E. Williams, J. Zobel, and D. Bahle. Fast Phrase Querying with Combined Indexes. *ACM Transactions on Information Systems*, 22(4):573–594, 2004.

[38] J. Zobel and A. Moffat. Inverted Files for Text Search Engines. *ACM Computing Surveys*, 38(2), 2006.