Compressed Self-Indices Supporting Conjunctive Queries on Document Collections

Diego Arroyuelo[1], Senén González[2], and Mauricio Oyarzún[3]

[1] Yahoo! Research Latin America.
Blanco Encalada 2120, Santiago, Chile.
darroyue@yahoo-inc.com
[2] Department of Computer Science, Universidad de Chile.
sgonzale@dcc.uchile.cl
[3] Universidad de Santiago de Chile.
mauricio.silvaoy@usach.cl

Abstract. We prove that a document collection, represented as a unique sequence T of n terms over a vocabulary Σ, can be represented in $nH_0(T) + o(n)(H_0(T) + 1)$ bits of space, such that a conjunctive query $t_1 \wedge \cdots \wedge t_k$ can be answered in $O(k\delta \log\log |\Sigma|)$ adaptive time, where δ is the instance difficulty of the query, as defined by Barbay and Kenyon in their SODA'02 paper, and $H_0(T)$ is the empirical entropy of order 0 of T. As a comparison, using an inverted index plus the adaptive intersection algorithm by Barbay and Kenyon takes $O(k\delta \log \frac{n_M}{\delta})$, where n_M is the length of the shortest and longest occurrence lists, respectively, among those of the query terms. Thus, we can replace an inverted index by a more space-efficient in-memory encoding, outperforming the query performance of inverted indices when the ratio $\frac{n_M}{\delta}$ is $\omega(\log |\Sigma|)$.

1 Introduction, Results and Previous Work

Text retrieval systems allow the access to big text collections in order to retrieve information that satisfies the information needs of users, so they are fundamental for information retrieval (IR) [2]. Let $\mathcal{D} = \{D_1, \ldots, D_N\}$ be a collection of N documents, where each document D_i is modeled as a sequence of *terms* (or *words*) from a vocabulary Σ of size $|\Sigma|$. We assume that every original term in Σ has been assigned a unique term identifier. Thus, from now by "term" we will mean "term identifier". *Conjunctive queries* $t_1 \wedge \cdots \wedge t_k$, asking to report the documents that contain all the $t_1, \ldots, t_k \in \Sigma$, are one of the most common kinds of queries issued to text retrieval systems. We assume a model where all the documents that satisfy the query are retrieved (as opposed to a model where only the most relevant documents need to be found).

Text retrieval systems are usually based on *inverted indices* [2, 37], which consists of a *vocabulary table* (containing the $|\Sigma|$ distinct *terms* from the collection) and an *occurrence list* for every term $c \in \Sigma$, storing the identifiers of the documents that contain the term c. Conjunctive queries are supported by intersecting the occurrence lists of the query terms [2, 1, 16, 34]. However, the occurrence lists must be precomputed and stored, which requires considerable amounts of space. So these must be compressed, and usually stored on secondary storage [2], in both cases yielding a slowdown in query processing time.

Given the popularity of inverted indices, it is not a surprise that most of the work on reducing the space requirements in IR has focussed on compressing the occurrence lists of inverted indices. Among the vast literature on the topic, we can cite some recent relevant work [36, 34]. An alternative that has emerged in recent years, and that seeks to avoid the use of secondary storage when dealing with large volumes of data, is that of compressed/succinct data structures [26, 30]. Because of the continuously growing data repositories available from different sources, reducing the space used by an algorithm is fundamental for efficiency matters. Hence, in recent years there have been several studies on succinct and compressed data structures, achieving small and functional data structures. For instance, there are representations for general trees of n nodes using just $2n+o(n)$ bits, while supporting a complete set of operations on it in $O(1)$ time [8, 19, 33]. This is $\Theta(\log n)$ times smaller than the traditional representation. Thus, succinct data structures are not only space-efficient, but also versatile and functional.

There are also compressed data structures for full-text search [30], where all the occurrences of a pattern in a text must be reported. These are called compressed self-indices, which replace a text with a representation that uses space proportional to that of the compressed text, allows indexed searches over the text, and extracting any text snippet. These indices provide interesting trade-offs in practice [18], allowing one to replace the traditional suffix trees and suffix arrays in many scenarios. This is an active and mature area of research. Compressed data structures supporting operations rank and select [11, 20, 4, 3] are also fundamental. Given a sequence of symbols S, $\mathsf{rank}_c(S, i)$ counts the number of c's in $S[1..i]$, and $\mathsf{select}_c(S, j)$ yields the position of the j-th occurrence of c in S. Operation $\mathsf{access}(S, i)$, which yields $S[i]$, is also relevant to retrieve the indexed sequence. These operations are usually the core for more complex succinct data structures [30], and shall be central to our work. Succinct data structures have also been successfully used to represent graphs [17, 13], functions and permutations [6, 7], and to support document retrieval [32, 35, 25], among others.

However, the impact of succinct/compressed data structures has not been as strong as expected in the area of document reporting in text retrieval systems. As we said, most of the achievements to reduce the space requirements in IR are related to compressing the inverted indices. There are several solutions for the problem of document reporting [29, 32, 35, 21, 25]. However, none of these support IR queries, like conjunctive queries, which must be solved by first generating the occurrence lists of the search patterns, to then apply standard intersection algorithms on these lists. This is, however, non-efficient, since an inverted index has precomputed these lists, and only the price for the intersection must be paid for. We aim at performing conjunctive queries directly over the index, without generating the occurrence lists beforehand. The idea of emulating list intersection algorithms in this way was hinted at [7], though without details and not regarding document collections, but just single texts. Their representation is based on a data structure for permutations, which represents, somehow, the *wavelet tree* [24] of the text. Thus, they cannot profit from the use of more efficient representations for $\mathsf{rank}/\mathsf{select}$ (e.g., [3]).

Instead of storing the occurrence lists, the work [10] proposes to use a compressed index to generate them on the fly, thus saving considerable space (though increasing the query time). However, they index just single texts, and *all* the query occurrences need to be found. This can be adapted to represent document collections, though the total query time becomes proportional to the number of query occurrences, rather than the number of documents containing it.

Thus, there is little (or none) support in the compressed data structure literature for operations that are fundamental in IR. An important result in this track would be, for instance, being able to replace an inverted index by a compressed/succinct encoding, while still supporting efficient conjunctive queries.

Let $T[1..n]$ denote the sequence obtained from the concatenation of the documents in the collection. Our main contribution is an study on the support of conjunctive queries in self-indexed text retrieval systems. We show that any $\mathsf{rank}/\mathsf{select}$ data structure (supporting operation access in time $O(a)$, rank in $O(r)$ time, and select in $O(s)$ time) is powerful enough to support the following IR operations: (i) Extracting any document snippet of length ℓ in $O(a \cdot \ell)$ time; (ii) obtaining the occurrence list (of length occ) of a given query term in time $O((r+s)occ)$; and (iii) conjunctive queries $t_1 \wedge \cdots \wedge t_k$, for $k \geq 2$, in $O(k\delta(r+s))$ adaptive time, where δ is the instance difficulty of the query, as defined by [5].

In particular, we use the $\mathsf{rank}/\mathsf{select}$ data structure from [3], and achieve $nH_0(T)+o(n)(H_0(T)+1)$ bits of space, such that snippet extraction can be carried out in $O(\ell)$ time, and conjunctive queries can be answered in $O(k\delta \log\log|\Sigma|)$ adaptive time, where $H_0(T)$ is the empirical entropy of order 0 of T [28]. It is important to note that $\delta \leq n_m$, where n_m is the length of the shortest occurrence list among those of the query terms. As a comparison, the time achieved by an inverted index plus the adaptive intersection algorithm by Barbay and Kenyon [5] is $O(k\delta \log \frac{n_M}{\delta})$, where n_M is the length of the longest occurrence list among those of the query terms. Thus, we can replace an inverted index by a more space-efficient in-memory encoding, which outperforms inverted indices when the ratio n_M/δ is $\omega(\log|\Sigma|)$.

2 Preliminary Concepts and Notation

Succinct Data Structures for rank *and* select. Given a sequence $S[1..n]$ over an alphabet $\Sigma = \{1, \ldots, |\Sigma|\}$, operation $\mathsf{rank}_c(S, i)$, for $c \in \Sigma$, counts the number of c's occurring in $S[1..i]$. Operation $\mathsf{select}_c(S, j)$ is defined as the position of the j-th occurrence of c in S (we assume that select yields $n+1$ iff the number of c's in S is less than j). The representation by Barbay et al. [3] uses $nH_0(S) + o(n)(H_0(T)+1)$ bits, where $H_0(T) \leq \log|\Sigma|$ denotes the empirical entropy of order 0 of T [28]. Operations rank_c and select_c on S, for any $c \in \Sigma$, are supported in $O(\log\log|\Sigma|)$ time, as well as access in $O(1)$ time [3] . Let r and s denote the time complexities for the rank and select operations, respectively.

Our Representation for the Document Collection. Let $\mathcal{D} = \{D_1, \ldots, D_N\}$ be a *document collection* of size N, where each document is represented as a sequence $D_i[1..l_i]$ of l_i *terms* from a vocabulary Σ of size $|\Sigma|$. Assuming that '\$'$\notin \Sigma$ is a special separator symbol, we build the sequence:

$$T[1..n] = \$D_1\$D_2\$\cdots\$D_N\$$$

of length $n = 1 + \sum_{i=1}^{N}(l_i + 1)$ and size $n \log |\Sigma|$ bits. The order of the concatenation is arbitrary. A convenient order for document ranking purposes can be the one given by a global ranking function, such as Hits [27] or Pagerank [9].

Each document D_i is assigned a unique *document identifier* i. If we represent T with a rank/select data structure, then given any position $1 \leq j \leq n$, operation $\mathsf{get\_docid}(j) \equiv \mathsf{rank}_\$(T, j)$ yields the document identifier of the document j belongs to. Given a document identifier $1 \leq i \leq N$, one can obtain the starting position within T for document D_i as $\mathsf{get\_doc}(i) \equiv \mathsf{select}_\$(T, i) + 1$.

3 Succinct Encodings for Document Reporting

The work [10] showed that an instance of rank/select data structure (in particular, a *wavelet tree* [24]) is competitive with an inverted index for reporting *all* the occurrences of a query term t. This is relevant for text searching, where all the occurrences need to be found. In our case, however, we should search for every occurrence of t, and for each determine the document that contains it, which is reported (without repetitions). However, this is wasteful when there are many occurrences of t, but just a few documents actually contain it.

To work in time proportional to the number of documents containing the query term t, we locate the first occurrence of t within T by using $j = \mathsf{select}_t(T, 1)$. We compute next the document identifier $d = \mathsf{get\_docid}(j)$ of the document containing the term, and report it. Then, with $j = \mathsf{select}_\$(T, d + 1)$ we jump up to the end of the current document, and $f = \mathsf{rank}_t(T, j)$ counts the number of occurrences of t up to position j. Then, we jump to the next document containing an occurrence of t by means of $\mathsf{select}_t(T, f + 1)$, and repeat the procedure. The running time of this algorithm is $O((r+s)occ)$. By using the data structure from [3] we obtain $nH_0(T) + o(n)(H_0(T) + 1)$ bits of space, while the *occ* documents containing a query term can be computed in $O(occ \log\log |\Sigma|)$ time

4 Efficient Support for Conjunctive Queries

Traditionally, conjunctive queries are supported by intersecting the occurrence lists of the individual query terms [2, 1, 16, 34]. If the document collection has been encoded as in Section 3, a simple solution for a query $t_1 \wedge t_2$ could be to generate the occurrence lists $\mathtt{Occ}_1[1..n_1]$ and $\mathtt{Occ}_2[1..n_2]$ for t_1 and t_2, respectively, and then intersect them by using any intersection algorithm [1, 16, 34]. The lower bound for the intersection problem in the comparison model is $\Omega(n_1 \log \frac{n_2}{n_1})$ time, assuming that $n_1 \leq n_2$, which is achieved by the work of Baeza-Yates [1]. Thus, the time for conjunctive queries would be $O((r+s)(n_1+n_2)+n_1\log\frac{n_2}{n_1})$ in the worst case. For $k>1$, this time is $O((r+s)\sum_{i=1}^{k} n_i + kn_m\log\frac{n_M}{n_m})$, where n_i denotes the length of the occurrence list of t_i, and n_m and n_M denote the lengths of the shortest and longest occurrence lists, respectively. By using an adaptive intersection algorithm [15, 5], the time would be $O((r+s)\sum_{i=1}^{k} n_i + k\delta\log\frac{n_M}{\delta})$, where $\delta \le n_m$ is the difficulty of the instance as defined by [5].

Thus, generating the occurrence lists before the intersection phase is wasteful, since not every occurrence of t_i's is useful for the intersection. Our aim is to avoid this cost, so we can conceptually think of our indices as storing the lists.

4.1 A Simple Worst-Case Algorithm for Conjunctive Queries

Given a query $t_1 \wedge t_2$, a first approach to reduce the query cost is to obtain the occurrence list $\mathtt{Occ}_1[1..n_1]$ for t_1 as in Section 3. Then, for every document $\mathtt{Occ}_1[i]$ containing t_1, we check whether t_2 occurs within it, which is true iff:

$$\mathsf{rank}_{t_2}(T,\mathsf{get\_doc}(\mathtt{Occ}_1[i]+1)-1)-\mathsf{rank}_{t_2}(T,\mathsf{get\_doc}(\mathtt{Occ}_1[i]))>0. \quad (1)$$

We work in time $O((r+s)n_1)$, thus saving the time to generate the list of t_2. To minimize the time, we should generate first the shortest occurrence list. However, we must store the length of the occurrence list of every term, which uses $|\Sigma|\log N$ extra bits of space, and is usually negligible in practice.

For queries of the form $t_1 \wedge \cdots \wedge t_k$, we first sort the query terms by their number of occurrences, then generate the occurrence list for the less frequent term, and then use it to drive the candidate checking, considering one term at a time in the order given by the sorting. Thus, the total time is $O(k\log k + k(r+s)n_m)$. By using the data structure from [3], we obtain:

Theorem 1. *Given a document collection T represented as a sequence of n terms over a vocabulary Σ, it can be replaced by a representation that uses $nH_0(T)+o(n)(H_0(T)+1)$ bits of space, supports extracting any document snippet of length ℓ in $O(\ell)$ time, and finding all the documents that answer a query $t_1 \wedge \cdots \wedge t_k$ in $O(k\log k + kn_m\log\log|\Sigma|)$ worst-case time, where n_m denotes the length of the shortest occurrence list among those of the query terms.*

4.2 An Adaptive Algorithm for Conjunctive Queries

Instead of performing a pair-wise intersection (as with the previous algorithm), this time we search for all query terms at once, in some sense carrying out the "intersection" as we search for them.

For a query $t_1 \wedge t_2$, we search for the first occurrence of each of the query terms, $s_1 = \mathsf{select}_{t_1}(T,1)$ and $s_2 = \mathsf{select}_{t_2}(T,1)$. Assume, without loss of generality, that $s_1 < s_2$ and that these correspond to document identifiers $d_i < d_{i+m}$, respectively, for $m \ge 1$. Then, notice that it is not necessary to search for the occurrences of t_1 in documents $d_{i+1},\ldots,d_{i+m-1}$, since there is no occurrence of t_2 within them. Thus, we move s_1 up to the starting position of document

d_{i+m}, and search for the next occurrence of t_1 from there (by using select). If this lies within document d_m, then we report it as an occurrence. If the document identifier obtained is greater than d_m, then we move s_2 to the beginning of the document containing t_1, and repeat the procedure, moving forward through the text, until either s_1 or s_2 reach the value $n+1$ (recall that select yields $n+1$ when there are not enough occurrences).

Let n_1 and n_2 be the number of documents containing t_1 and t_2, respectively, and let $n_m = \min\{n_1, n_2\}$. It is not hard to see that the shortest occurrence list will be exhausted after carrying out at most $n_m + 1$ steps. This indicates that this algorithm works in $O((r+s)n_m)$ time in the worst case. An instance that yields such a behavior is as follows:

$$T = \$ \cdots | \cdots t_1 \cdots | \cdots t_2 \cdots | \cdots t_1 \cdots | \cdots t_2 \cdots | \cdots t_1 \cdots | \cdots t_2 \cdots | \cdots t_1 \cdots \$$$

where the original separator '\$' has been replaced by '|' for clarity (except for the first and last '\$'). In this example, we work in time proportional to n_2, since it is smaller. Moreover, this method can profit from the distribution of the query terms across T. For example, if we have an instance like this:

$$T = \$ \cdots | \cdots t_1 \cdots | \cdots | \cdots t_1 \cdots | \cdots | \cdots t_1 \cdots | \cdots \cdots | \cdots t_2 \cdots | \cdots | \cdots t_2 \cdots \$$$

and assuming that these are the only occurrences of t_1 and t_2 in T, it only takes $O(1)$ steps to determine that the result of the conjunctive query is empty. Notice the analogy with an integer intersection algorithm: this corresponds to the case where all values stored in the occurrence list of t_1 are smaller than those in the occurrence list of t_2. So our algorithm adapts nicely to this kind of instance.

In general, this algorithm is adaptive to this interleaving of the query terms within the document collection. For example, suppose that t_1 and t_2 occur within T in exactly δ of these groups. Then, notice that we need $O(\delta)$ steps to certify the result of the intersection. Though seen from a different perspective, notice that this is the same instance difficulty measure as the one introduced in [5], and also that $\delta \leq n_m$. Thus, the adaptive complexity of our algorithm is $O(\delta(r+s))$, which is $O((r+s)n_m)$ in the worst case.

This procedure can be generalized to any $k > 1$. Now, every query term t_i has an index s_i, which indicates the last occurrence of t_i that has been regarded. First, we look for $s_1 = \mathsf{select}_{t_1}(T, 1)$, and determine the document identifier $j_1 = \mathsf{get\_docid}(s_1)$ that contains it. Then, we look for the next occurrence of t_2 starting from document j_1. That is, let $j' = \mathsf{get\_doc}(j_1)$. Hence, we set $s_2 = \mathsf{select}_{t_2}(T, \mathsf{rank}_{t_2}(T, j') + 1)$. Then, let $j_2 = \mathsf{get\_docid}(s_2)$. So, we search for the next occurrence of t_3 starting from document j_2, and so on, regarding the query terms in a round-robin fashion. The process finishes when some s_i reaches $n+1$, since the corresponding list has been exhausted.

The adaptive analysis is similar to that for binary conjunctive queries. Hence, general conjunctive queries take $O(k\delta(r+s))$ time, where $\delta \leq n_m$ is our difficulty measure, as defined above. Thus, by using the data structure from [3], we obtain:

Theorem 2. *Given a document collection T represented as a sequence of n terms from an alphabet of size $|\Sigma|$, it can be replaced by a representation that* *uses $nH_0(T) + o(n)(H_0(T) + 1)$ bits of space, supports extracting any document snippet of length ℓ in $O(\ell)$ time, and finding all the documents that answer a query instance $t_1 \wedge \cdots \wedge t_k$ of difficulty $\delta \leq n_m$ in time $O(k\delta \log\log |\Sigma|)$, where n_m is the length of the shortest occurrence list among those of the query terms.*

5 Experimental Results

For our experimental results, we used a sample of 277,371 documents (taken at random) from the UK Web, collected by Yahoo! in 2006. This requires about 1.1 gigabytes, with a vocabulary of about 1.6 million terms. We used a query log of about 36 million queries submitted to `www.uk.yahoo.com` during three months of year 2006. Our computer is an Intel(R) Core(TM)2 Duo CPU at 2.80GHz, with 5 GB of RAM, and running version `2.6.31-20-server` of Linux kernel.

We decided to use a Huffman-shaped wavelet tree as a particular rank/select data structure, since these have proven to be a competitive choice in practice [12]. This uses $n(H_0(T) + 1) + o(n(H_0(T) + 1))$ bits. We based on the wavelet tree implementation from the `libcds` library, available in *Google code* [4]. Since our algorithms need an intensive use of select, we use the `darray` data structure from [31] [5] to represent the internal wavelet tree nodes. We modified the original `darray` implementation to reduce the overall space wasted, obtaining a wavelet tree that uses about 521 MB. The space achieved by representing the nodes with the very space-efficient data structure from [22] is about 500 MB. However, by using `darray` the times become around 4–5 times faster.

The first step of the algorithm from Section 4.1 generates the occurrence list $\mathtt{Occ}_1[1..n_1]$ for term t_1. Then, for every remaining query term we check with Eq. (1) which of the documents in $\mathtt{Occ}_1$ contain it. However, this involves the use of select. Let $\mathtt{DocBegin}[1..N]$ be a table storing the document beginnings in $N \log n$ extra bits. An equivalent test, though faster in practice, is:

$$\mathsf{rank}_{t_2}(T, \mathtt{DocBegin}[\mathtt{Occ}[i] + 1] - 1) - \mathsf{rank}_{t_2}(T, \mathtt{DocBegin}[\mathtt{Occ}[i]]) > 0, \qquad (2)$$

Hence, the selects in this algorithm come from generating the occurrence list $\mathtt{Occ}_1$, which amount to n_1 selects. The remainder works with rank.

In our experiments, we call `SLF` (from shortest-list first) the algorithm from Section 4.1, and `Adaptive` the one from Section 4.2. We searched for 1 million random queries from our query log, most of them of length from 1 to 6. We show in Table 1 the number of queries answered per second, for a random (top part) and a non-random (bottom part) ordering of the documents. In both cases, we show experimental results for queries that return a non-empty answer, as well as for queries that not necessarily have a non-empty answer. The non-random ordering aims at showing the adaptability of `Adaptive`. We use the following simple heuristic: we first take the vocabulary of the different terms from our

[4] We thank Francisco Claude for providing the source code for the `libcds` library.

[5] We thank Kunihiko Sadakane for providing the source code for the `darray` data structure.

query log, and construct an inverted index of the document collection for these query terms. We then take the occurrence list for the most frequent term (i.e., the longest list), and concatenate the documents that appear in the list. We proceed in the same way with the remaining occurrence lists, in the order given by their lengths (we avoid duplicate documents when performing the concatenation).

We also implemented an inverted index by representing the occurrence lists by their differences, and for every list we use the exact number of bits to represent the highest difference in the list. This allows us a much better decompression performance, while achieving an acceptable compression ratio: 189 MB, which is slightly more space than that used, for instance, by a Golomb-compressed inverted index. However, as we represent the lists by differences, we cannot apply binary/doubling search on them, so we cannot use efficient intersection algorithms like the ones at [1, 5]. Hence, we use a two-level approach similar to the one at [14]. We sample one out of B values in the list, and represent them in absolute way. Thus, binary search can be used on them, which is followed by a sequential search on the corresponding block of the list. We chose a typical value $B = 8$, so we get an index that requires about 270 MB (recall that this does not include the text). It is important to note that we only store document identifiers in the occurrence lists. No extra information about term frequencies, positional information, or any other information for ranking, is stored in the lists.

As it can be seen, and unlike `Adaptive`, the inverted index and `SLF` are insensitive to the document ordering, as expected. It is important to note how the intensive use of `rank` (rather than `select`) in `SLF` yields in all cases a (sometimes slightly) better performance than `Adaptive`. We think that by using a better heuristic to concatenate the documents we can get better results.

As a comparison, rather than using `SLF` or `Adaptive` to compute the answer, we used the algorithm from Section 3 to generate the occurrence lists of each query term, yet without using any intersection algorithm afterwards to compute the final answer. We were able, in this way, to answer less than 10 queries per second. This indicates that, independently of the intersection algorithm used on the lists, this approach will be outperformed by ours. Thus, saving the time to generate the lists is very important.

When comparing with the inverted index, we conclude that our algorithms can be up to 5–7 times slower, while using about 1.92 times the space of the inverted index. However, the latter does not include the text, so snippet extraction is not possible. To achieve this, we must add the text to the index, for instance represented in compressed for with a wavelet tree (this would support extracting arbitrary text snippets). This would add about 500–520 MB to the inverted index, for a total space usage of about 1.5 times the space of our algorithms. Also, our algorithms store more information than the text itself. For instance, we can use `rank` to compute the frequency of a term within a given document. We can also know the positions of a term within a document, allowing this kind of information to be used in more involved ranking functions.

In the above experiments, algorithm `SLF` outperforms `Adaptive` in many cases. However, for queries with $k > 1$, we made the experiment of searching

Table 1. Number of queries answered per second, for a random (top) and non-random (bottom) ordering of the document collection.

	`SLF` (521 MB)	`Adaptive` (521 MB)	`Hybrid` (521 MB)	Inverted Index (270 MB, **no text**)
Every query has an answer	74	58	68	381
Not every query has an answer	102	82	95	583
Every query has an answer	77	74	82	381
Not every query has an answer	106	102	115	583

just for the two terms having the smallest occurrence lists. The result is that `Adaptive` is faster than `SLF` for doing this, as it can be seen in Fig. 1 (left), assuming that the document ordering explained above has been used. This is because the performance of `Adaptive` depends on the interleave factor δ. When we search for long queries, this factor can be high (close to n_{min}), because of the many terms that compose the query. However, if we search just for the two terms having the smallest lists, we minimize the probability of interleaving. Hence, `Adaptive` outperforms `SLF`. This fact improves with the query length, as it can be seen in Fig. 1 (left), which is explained by the result in Fig. 1 (right), which compares the length of the two shortest list for different query lengths. When searching for three terms, instead of two, we concluded that `SLF` outperforms `Adaptive`.

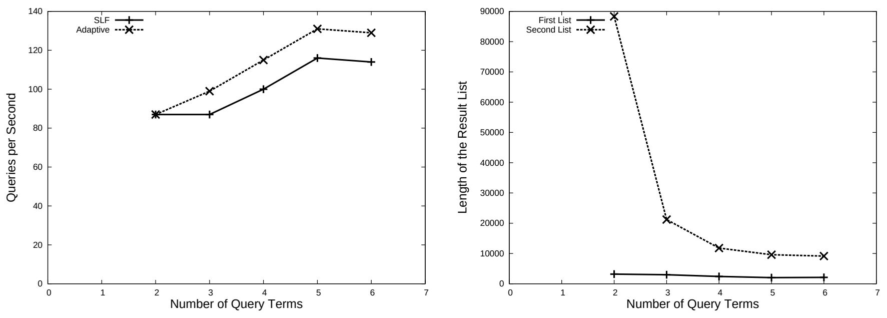
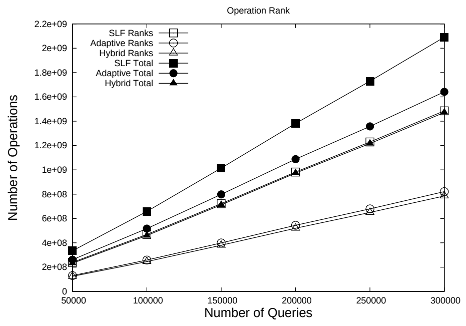


Fig. 1. Experimental time for searching the two terms with the shortest lists, for queries of different length (left); comparison of the length of the two shortest lists (right).

We use the above fact to define a hybrid scheme (called `Hybrid`) that first searches for the two terms with shortest lists using algorithm `Adaptive`, and then use this partial result to search with `SLF`. The result is shown in Table 1. As it can be seen, we obtain an improvement of about 10% in the query performance.

Finally, in Fig. 2 we show the performance of our algorithms from another point of view: the number of rank/select operations needed to compute an answer. As it can be seen, the total number of operations (i.e., number of ranks



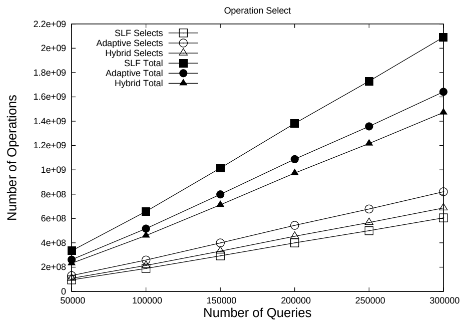


Fig. 2. Number of rank (left) and select (right) operations performed by our algorithms.

plus number of selects) performed by algorithm SLF is about 1.27 times the number of operations of Adaptive, and 1.43 times the number of operations of Hybrid. However, according to our experiments, SLF outperforms the others when comparing the total time. This can be explained by the fact that most of the operations of SLF are just ranks (71%), which are supported in a much faster way in the wavelet tree implementation that we use [12]. Thus, our experimental results from Table 1 are clearly dependent on the rank/select representation used. A representation with a more efficient support for select would produce much better results for the Adaptive scheme. For instance, the data structure from [3] is able to support select in $O(1)$ time, so this could be relevant for our purposes. This representation could be also useful to compete against the inverted indices.

6 Conclusions

We proved that any rank/select data structures is powerful enough so as to represent a document collection using $nH_0(T) + o(n)(H_0(T) + 1)$ bits of space and support IR operations on it. Thus, a conjunctive query $t_1 \wedge \cdots \wedge t_k$ can be answered in $O(k\delta \log\log|\Sigma|)$ adaptive time, where δ is the instance difficulty of the query [5], and $H_0(T)$ is the empirical entropy of order 0 of the collection. This outperforms the intersection algorithm [5] when the ratio $\frac{n_M}{\delta}$ is $\omega(\log|\Sigma|)$.

To conclude, our algorithms are powerful and simple to implement. The former is, on the one hand, because in theory we can perform as efficiently as (and in many cases even better than) the most efficient algorithms over inverted indices. The latter is, on the other hand, because we do not introduce any complicated data structure to support the operations, but rather any rank/select data structure can be used [12]. Our experimental results show that our result are promising. However, further work need to be done in order to obtain a query performance similar to that of inverted indices. We hope to achieve this by using a faster implementation for select, as for instance the one provided by [3].

We did not regard dynamic document collections in this paper (that is, document collections where documents are added and deleted, as well as modified). However, our algorithms can be also supported in such a case by using appropriate dynamic data structures for rank and select [23].

Acknowledgments. We thank Mauricio Marin and Gonzalo Navarro for many fruitful discussions, suggestions, and proofreading an earlier version of this paper.

References

1. R. Baeza-Yates. A fast set intersection algorithm for sorted sequences. In *Proc. CPM*, LNCS 3109, pages 400–408, 2004.
2. R. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval.* ACM Press / Addison-Wesley, 1999.
3. J. Barbay, T. Gagie, G. Navarro, and Y. Nekrich. Alphabet partitioning for compressed rank/select with applications. *CoRR*, abs/0911.4981, 2009.
4. J. Barbay, M. He, J. I. Munro, and S. S. Rao. Succinct indexes for strings, binary relations and multi-labeled trees. In *Proc. of SODA*, pages 680–689, 2007.
5. J. Barbay and C. Kenyon. Adaptive intersection and t-threshold problems. In *SODA*, pages 390–399, 2002.
6. J. Barbay and J. I. Munro. Succinct encoding of permutations: Applications to text indexing. In M.-Y. Kao, editor, *Encyclopedia of Algorithms.* Springer, 2008.
7. J. Barbay and G. Navarro. Compressed representations of permutations, and applications. In *Proc. STACS*, pages 111–122, 2009.
8. D. Benoit, E. Demaine, J. I. Munro, R. Raman, V. Raman, and S. S. Rao. Representing trees of higher degree. *Algorithmica*, 43(4):275–292, 2005.
9. S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. *Computer Networks*, 30(1-7):107–117, 1998.
10. N. Brisaboa, A. Fariña, S. Ladra, and G. Navarro. Reorganizing compressed text. In *Proc. SIGIR*, pages 139–146, 2008.
11. D. Clark and J. I. Munro. Efficient suffix trees on secondary storage. In *Proc. SODA*, pages 383–391, 1996.
12. F. Claude and G. Navarro. Practical rank/select queries over arbitrary sequences. In *Proc. SPIRE*, LNCS 5280, pages 176–187. Springer, 2008.
13. F. Claude and G. Navarro. Extended compact web graph representations. In *Algorithms and Applications*, LNCS 6060, pages 77–91, 2010.
14. J. S. Culpepper and A. Moffat. Compact set representation for information retrieval. In *SPIRE*, pages 137–148, 2007.
15. E. Demaine, A. López-Ortiz, and J. I. Munro. Adaptive set intersections, unions, and differences. In *SODA*, pages 743–752, 2000.
16. E. Demaine, A. López-Ortiz, and J. I. Munro. Experiments on adaptive set intersections for text retrieval systems. In *ALENEX*, pages 91–104, 2001.
17. A. Farzan and J. I. Munro. Succinct representations of arbitrary graphs. In *ESA*, LNCS 5193, pages 393–404, 2008.
18. P. Ferragina, R. González, G. Navarro, and R. Venturini. Compressed text indexes: From theory to practice. *ACM Journal of Experimental Algorithmics*, 13, 2008.
19. P. Ferragina, F. Luccio, G. Manzini, and S. Muthukrishnan. Compressing and indexing labeled trees, with applications. *Journal of the ACM*, 57(1), 2009.

20. P. Ferragina, G. Manzini, V. Mäkinen, and G. Navarro. Compressed representations of sequences and full-text indexes. *ACM TALG*, 3(2):article 20, 2007.
21. T. Gagie, S. Puglisi, and A. Turpin. Range quantile queries: Another virtue of wavelet trees. In *SPIRE*, LNCS 5721, pages 1–6, 2009.
22. R. González, Sz. Grabowski, V. Mäkinen, and G. Navarro. Practical implementation of rank and select queries. In *Poster Proc. of WEA*, pages 27–38, 2005.
23. R. González and G. Navarro. Rank/select on dynamic compressed sequences and applications. *Theoretical Computer Science*, 410:4414–4422, 2008.
24. R. Grossi, A. Gupta, and J. S. Vitter. High-order entropy-compressed text indexes. In *Proc. SODA*, pages 841–850, 2003.
25. W.-K. Hon, R. Shah, and J. S. Vitter. Space-efficient framework for top-k string retrieval problems. In *FOCS*, pages 713–722, 2009.
26. G. Jacobson. *Succinct static data structures.* PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, 1988.
27. J. M. Kleinberg. Authoritative sources in a hyperlinked environment. *J. ACM*, 46(5):604–632, 1999.
28. G. Manzini. An analysis of the Burrows-Wheeler transform. *J. ACM*, 48(3):407–430, 2001.
29. S. Muthukrishnan. Efficient algorithms for document retrieval problems. In *SODA*, pages 657–666, 2002.
30. G. Navarro and V. Mäkinen. Compressed full-text indexes. *ACM Computing Surveys*, 39(1):article 2, 2007.
31. D. Okanohara and K. Sadakane. Practical entropy-compressed rank/select dictionary. In *Proc. ALENEX*, pages 60–70, 2007.
32. K. Sadakane. Succinct data structures for flexible text retrieval systems. *J. Discrete Algorithms*, 5(1):12–22, 2007.
33. K. Sadakane and G. Navarro. Fully-functional succinct trees. In *Proc. SODA*, pages 134–149, 2010.
34. P. Sanders and F. Transier. Intersection in integer inverted indices. In *ALENEX*, 2007.
35. N. Välimäki and V. Mäkinen. Space-efficient algorithms for document retrieval. In *CPM*, LNCS 4580, pages 205–215, 2007.
36. H. Yan, S. Ding, and T. Suel. Inverted index compression and query processing with optimized document ordering. In *Proc. WWW*, pages 401–410, 2009.
37. J. Zobel and A. Moffat. Inverted files for text search engines. *ACM Comput. Surv.*, 38(2), 2006.