Indexes for Highly Repetitive Document Collections

Francisco Claude
School of Computer Science,
University of Waterloo
Waterloo, Canada
fclaude@cs.uwaterloo.ca

Antonio Fariña
Database Lab.,
University of A Coruña,
A Coruña, Spain.
fari@udc.es

Gonzalo Navarro
Dept. of Computer Science,
University of Chile,
Santiago, Chile.
gnavarro@dcc.uchile.cl

Miguel A. Martínez-Prieto
Dept. of Computer Science,
University of Valladolid,
Valladolid, Spain.
Dept. of Computer Science,
University of Chile,
Santiago, Chile,
migumar2@infor.uva.es

ABSTRACT

We introduce new compressed inverted indexes for highly repetitive document collections. They are based on run-length, Lempel-Ziv, or grammar-based compression of the differential inverted lists, instead of gap-encoding them as is the usual practice. We show that our compression methods significantly reduce the space achieved by classical compression, at the price of moderate slowdowns. Moreover, many of our methods are universal, that is, they do not need to know the versioning structure of the collection.

We also introduce compressed self-indexes in the comparison. We show that techniques can compress much further, using a small fraction of the space required by our new inverted indexes, yet they are orders of magnitude slower.

Categories and Subject Descriptors

E.1 [**Data Structures**]; E.4 [**Coding and Information Theory**]; H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval

General Terms

Algorithms

1. INTRODUCTION

Large versioned document collections, such as *Wikipedia* (`www.wikipedia.org`) and the *Wayback Machine* from the *Internet Archive* (`www.archive.org/web/web.php`), are examples of the emergence of highly repetitive document collections, where most documents are near-duplicates. Apart from versioned document collections, other cases where this problem arises are software repositories (where the tree of versions is maintained), biological databases (where many DNA or protein sequences of the same or related species are maintained), periodic technical publications (where the same data, with small updates, are republished), and so on.

These collections may be very large, but at the same time highly compressible. While Lempel-Ziv compressors [24] are successful in capturing their repetitiveness, such compression is suitable only for archival purposes. The applications we have enumerated require, instead, fast searching and direct access capabilities. Thus, we need to compress not only the data, but also the indexes.

There is a burst of recent activity in exploiting this repetitiveness at the indexing structures in order to provide fast searches into the collection within little space. Both inverted indexes for word and phrase queries over natural language texts [2, 5, 11, 12], and other indexes for general string collections [16, 6, 14, 7], have been pursued.

The focus of this paper is on natural language text collections, which can be decomposed into words, and queried for words or phrases. The classical data structure to index such collections is the inverted index, where a list of the occurrences of each distinct word is maintained. The variant where the lists are sorted by increasing document identifier has gained relevance, since such ordering is most useful for list intersections. Intersections of inverted lists arise as a fundamental task under the Google-like default policy of treating bag-of-word queries as ranked AND-queries. Therefore, intersections conform the heaviest part of the search process, and relevance ranking is done as a postprocessing step [9]. Intersections are also used for phrase queries.

In this context, there are two different types of indexes. *Non-positional* indexes find, given a word or bag-of-words query, the documents containing all the words. They store, for each word, the increasing list of documents containing it. *Positional* indexes retrieve the precise positions in each document where a word or phrase query appears. They store, in addition to the document identifiers, the word offset of the occurrences within each document.

Traditional techniques to compress inverted indexes [25] represent the differences between consecutive document or position values. Many of those differences tend to be small, and thus they are encoded in a way that favors small numbers. While very effective for traditional collections, this compression technique fails to capture the repetitiveness that arises in versioned collections.

He et al. [11, 12] have presented alternative compression

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
CIKM'11, October 24–28, 2011, Glasgow, Scotland, UK.
Copyright 2011 ACM 978-1-4503-0717-8/11/10 ...$5.00.

methods specifically targeted at highly repetitive collections. Their approach merges all versions of each document for creating the inverted lists and then keeps a secondary index that allows one to list the versions of a document that contain a given term. They obtained spectacular compression results for non-positional inverted indexes.

In this paper we introduce techniques for compressing inverted indexes on highly repetitive collections. Some of those are *universal* in the sense that they do not need to identify which document is a version of which. The techniques of He et al. [11, 12] work under a model where there exists a set of independent documents, each of which has a number of versions, and this versioning information must be available to the index. Our techniques can also work on cases where the versions form a tree structure (as in collaborative document creation, software repositories, or phylogenetic trees), or where it is unknown or unclear which documents are versions of which (as in DNA sequence databases).

Instead of using the classical encoding of differences, we adapt and apply run-length, grammar [15] or Lempel-Ziv [24] compression on the lists of differences. Our techniques largely outperform classical inverted index compression in this scenario. For example, our Lempel-Ziv-based index is 15 times smaller than a Rice-encoded index (the best choice for classical text collections), and at most 70% slower on word and conjunctive queries. Our grammar-based compressed index is up to 18 times smaller and up to 3 times slower. Our experiments also show that other classical encodings, such as Simple9 [1] and PforDelta [26], perform surprisingly well on repetitive collections, yet they still require 5 times more space than ours. Our techniques still do not match the performance of He et al.'s methods [12] when their assumptions hold, but these methods are not universal.

Some of our methods work well also for positional indexes, where the work of He et al. does not apply but other techniques are possible. Our Lempel-Ziv based index is 3.6 times smaller, yet up to 8 times slower, than classical inverted indexes. For this case we also consider self-indexes, which are compressed indexes (not based on inverted indexes) that encompass the text and the index. We use and adapt existing self-indexes designed for highly repetitive collections (mostly for computational biology scenarios) [16, 6, 14]. They compress up to 16 times more than our Lempel-Ziv-based index, yet their query times are 10 to 500 times higher.

2. RELATED WORK

2.1 Data structures for inverted lists

The compression of inverted lists usually represents each list $\langle p_1, p_2, p_3, \ldots, p_\ell \rangle$ as a sequence of d-gaps $\langle p_1, p_2 - p_1, p_3 - p_2, \ldots, p_\ell - p_{\ell-1} \rangle$, and uses a variable-length encoding for these differences, such as Rice codes [25]. Those methods assign shorter codes to smaller values, taking advantage of the fact that, on longer lists, the d-gaps are shorter.

If the lists reside on disk, reducing their space (and then the I/O cost to fetch them) is the most important criterion. If they reside in RAM, we must also consider the time to traverse them. We pay attention to both in this paper.

Gap encoding techniques that allow very fast in-memory traversals are `Vbyte` [23], `Simple9` [1] and `PforDelta` [26].

Intersections can be carried out by traversing the lists sequentially. When one list is much shorter than the other, it is advantageous to provide direct access so that the longer list can be searched for the elements of the shorter one. It was shown experimentally [3] that in practice the best is to sort the lists by length, taking the shortest as the "candidate" list, and iteratively intersect the candidate list with longer and longer lists, making it shorter and shorter. A technique to provide random access [8] samples regularly the compressed list and stores separately the array of samples, which is searched with exponential search. Very long lists are replaced by a bitmap. Given a parameter k, a list of length ℓ is sampled every $k \log_2 \ell$ positions. Another good method [22] regularly samples the universe of positions, so that the exponential search is avoided. Given a parameter B, it samples the universe of size u at intervals $2^{\lceil \log_2(uB/\ell) \rceil}$.

In the particular case of highly repetitive collections, the best figures so far have been presented by He et al. [12] in the non-positional case. They model versioned document collections using so-called *two-level indexes*. The first level of the inverted list of a term stores all the documents where the term appears in, at least, one of its versions. The second level is a bit vector that marks all document versions where the term appears. Experiments on Wikipedia and Internet Archive subsets report improvements in space and query times with respect to the original one-level techniques.

2.2 Re-Pair compression algorithm

Re-Pair [15] is a linear-time compressor that repeatedly finds the most frequent pair of symbols in a sequence of integers and replaces it with a new symbol, until no more replacements are useful. Over a sequence L, Re-Pair: (1) Identifies the most frequent pair ab in L; (2) Adds the rule $s \rightarrow ab$ to a dictionary R, where s is a new symbol not appearing in L; (3) Replaces every occurrence of ab in L by s; (4) Iterates until every pair in L appears once.

We call C the sequence resulting from L after compression. Every symbol in C represents a *phrase* (a substring of L), which is of length 1 if it is an original symbol (called a *terminal*) or longer if it is an introduced one (a *non-terminal*). Any phrase can be recursively expanded in optimal time (i.e., proportional to its length), even if C is stored on secondary memory (as long as the rules R fit in RAM).

We represent R using a method [10] that allows accessing any rule without decompressing the whole set of rules. It represents the DAG of rules as a set of trees. Each tree is represented as a sequence of leaf values (collected into a sequence R_S) and a bitmap that defines the tree shapes in preorder (collected into a bitmap R_B). Nonterminals are represented by the starting position of their tree (or subtree) in R_B. In R_B, internal nodes are represented by 1s and leaves by 0s, so that the value of the leaf at position i in R_B is found at $R_S[rank_0(R_B, i)]$. Operation $rank_0$ counts the number of 0s in $R_B[1, i]$ and can be implemented in constant time, after a linear-time preprocessing that stores only $o(|R_B|)$ bits of space [18] on top of the bitmap. To expand a nonterminal, we traverse R_B and extract the leaf values, until we have seen more 0s than 1s. Leaf values corresponding to nonterminals must be recursively expanded.

2.3 Lempel-Ziv compression

The Lempel-Ziv 1977 (*LZ77*) compression algorithm [24] works by decomposing the text from left to right into maximal phrases, so that each phrase is a substring of the previous text. The main drawback of LZ77 is that it does not support random access to the compressed text.

An alternative parsing called *LZ-End* [13] limits the substrings to end at a previous phrase, and uses this limitation to offer reasonable access time. This approach was shown [13] to compete very closely in space with LZ77.

2.4 Compressed self-indexes

Compressed self-indexes are data structures that enable efficient searches over an arbitrary string collection (called the text), and also replace the text by supporting extraction of arbitrary snippets or documents. The supported searches obtain all the positions of a substring in the collection, which makes them candidates to compete in the positional setting.

Self-indexes have undergone much progress in the last decade [19]. Recently they have been adapted to index highly repetitive sequences [16, 6, 14]. While general self-indexes have been successful by targeting at statistical compression, they have been proved insufficient on highly repetitive collections [16]. The indexes aimed at repetitive collections seek instead to capture repetitions in the text.

The first self-index successfully capturing high repetitiveness was the `RLCSA` [16]. This index adapts the well-known `CSA` of Sadakane [21] to better cope with the regularities that arise when indexing highly repetitive sequences. A variant aimed at indexing natural language, `WCSA` [4], regards the text as a sequence of words and separators instead of characters, but it is not so oriented to much repetitiveness. Another index aimed at repetitive collections is the `SLP` [7, 6], which exploits the regularities of highly repetitive sequences because its structure is determined by a grammar-based compressor (Re-Pair). We adapt the `SLP` to words (`WSLP`) in this paper. Finally, other strong indexes for repetitive sequences are `LZ77-index` and `LZend-index` [14], which are based on the LZ77 or LZ-End compression algorithms, respectively. The LZ77 parsing is, at least, as powerful as any grammar representation [20], and thus, it is also a good candidate for highly repetitive sequences.

3. NEW LIST REPRESENTATIONS

Our basic idea, for both non-positional and positional indexes, is to differentially encode the inverted lists, transforming a sequence $\langle p_1, p_2, p_3, \ldots, p_\ell \rangle$ into the d-gap sequence $\langle p_1, p_2 - p_1, p_3 - p_2, \ldots, p_\ell - p_{\ell-1} \rangle$, and then apply a general compression algorithm to the sequence formed by the concatenation of all the lists. Each vocabulary word will store a pointer to the beginning of its list in the compressed data.

3.1 Using run-length compression

A typical regularity in highly repetitive collections arises when the versions of a document happen to receive consecutive identifiers. As most of the words in such documents will appear in all versions, a consecutive sequence of numbers will appear in each inverted list of non-positional indexes. Consider the word w_t appearing in documents $d_i, \ldots, d_{i+k}$: the d-gapped list for w_t will contain $k-1$ consecutive ones.

In this representation we use any variable-length encoding for the differences. However, when this difference is 1, the next encoded number is the number of 1s in that run (k, in the previous paragraph). This encoding allows us skipping whole runs in a single operation, when processing intersections. It also allows combining with sampling techniques that support intersection methods other than the sequential one [8, 22]. In this paper we combine run-length compression with Rice coding, giving rise to method `Rice-Runs`.

Note that this technique works well only under the assumption that the documents can be linearized so that close documents receive consecutive numbers. While such kinds of assumptions are used in previous work [11, 12], we aim to handle more general cases in this paper. Moreover, this technique can be efficient only for non-positional indexes.

3.2 Using LZMA

This representation (already used for compressing q-gram indexes on DNA [6]) compresses each d-gap list with `Vbyte` and then with the *LZMA* variant of LZ77 (`www.7-zip.org`). LZMA is applied only on the lists where it reduces space. A bitmap marks which words were compressed with LZMA.

This representation, called `Vbyte-LZMA`, only supports extracting a list from the beginning, that is, we cannot jump to a random position on the list, and thus the only intersection algorithm supported is the sequential one. Moreover, unlike run-length compression, it cannot skip a compressed subsequence without fully processing it.

In this paper we apply LZMA to non-positional indexes as well, and evaluate it at intersections. LZMA handles more complex regularities than run-length compression. In particular, it works well on positional indexes: Consider r repetitions of a long substring S in the collection, for example in any r similar documents across the collection. For each word w_t in S, with occurrences at relative positions $i_1, i_2, \ldots, i_k$ in S, the sublist $i_2 - i_1, \ldots, i_k - i_{k-1}$ appears r times in the list of word w_t. Hence, LZMA will capture this repetition and represent $r-1$ of those sublists with just one reference. Note that this will occur independently of whether the versions are consecutive, and even without any need to know which documents are close versions of which.

4. RE-PAIR COMPRESSED LISTS

LZMA is limited to spotting intra-list regularities. A grammar-based compressor aimed at globally compressing the lists of d-gaps could spot inter-list similarities as well. In the same example of Section 3.2, consider that we have a phrase $w_{t_1} w_{t_2} \ldots w_{t_s}$ occurring k times in S. Then the sequence $i_2 - i_1, \ldots, i_k - i_{k-1}$ will appear r times inside each of the s lists, and a grammar compressor could be able to replace $rs-1$ of the occurrences by a single reference. LZMA would have replaced only $rs-s$.

We use Re-Pair as our grammar compressor. We operate over the integer values and not over their `Vbyte` encodings.

We prevent phrases from spanning multiple lists, which can be easily enforced at compression time. Thus we can store pointers from the vocabulary to the points in the compressed sequence C where the lists begin, and any list can be expanded in optimal time. We also store the Re-Pair dictionary, in the compact format described in Section 2.2.

The terminal symbols are directly the corresponding differential values (e.g. value *3* is represented by the terminal integer 3). This saves table accesses at decompression time.

4.1 Skipping

An attractive feature of our Re-Pair method is that we can add extra information to nonterminals that enables fast skipping over the compressed list data without decompressing. This yields much faster sequential list intersections.

The key idea is that *nonterminals also represent differential values*, namely the sum of the differences they expand into. We call this the *phrase sum* of the nonterminal.

Phrase sums will be stored in sequence R_S, aligned with the 1 bits of sequence R_B. Thus *rank* is not anymore necessary to move from one sequence to the other. The 0s in R_B are aligned in R_S to the leaf data, and the 1s to the phrase sums of the corresponding nonterminals.

In order to find whether a given document d is in the compressed list, we first scan the entries in C, adding up in a sum s the value $C[i]$ if it is a terminal, or $R_S[C[i]]$ if it is a nonterminal. If at some point we get $s = d$, then d is in the list. If instead $s > d$ at some point, we consider whether the last $C[i]$ processed is a terminal or not. If it is a terminal, then d is not in the list. If it is a nonterminal, we restart the process from $s - C[i]$ and process the R_S values corresponding to the 0s in $R_B[C[i], \ldots]$, recursing as necessary until we get $s = d$ or $s > d$ after reading a terminal.

4.2 Intersection algorithm

To intersect several lists, we sort them in increasing order of their *uncompressed* length (which we store separately). Then we proceed iteratively, searching in step i the list $i+1$ for the elements of the candidate list (recall Section 2.1).

At each intersection, the candidate list is sequentially traversed. Let x be its current element. We skip phrases of list $i + 1$, accumulating gaps until exceeding x, and then consider the previous and current cumulative gaps, $x_1 \leq x < x_2$. Then the last phrase represents the range $[x_1, x_2)$. If $x_1 = x$ we report x and shift to the next element in the candidate list. In either case, we advance in the candidate list until finding the largest $x' < x_2$. Then, we process all the interval $[x, x']$ within the phrase representing $[x_1, x_2)$ in R_S in a recursive fashion, until one of the two lists gets empty.

This procedure can be speeded up by adding samples, but we do not consider them in this paper. We will only consider a variant that stores skipping information on nonterminals (`RePair-Skip`), and one that does not (`RePair`).

5. SELF-INDEXES

Self-indexes support positional indexing. We will apply them on the concatenated text collection, $\mathcal{D}$. The self-index will find all the occurrences of the pattern, yet usually not in order, so a sorting post-processing is necessary.

Some self-indexes compared in this paper (see Section 2.4), are character-oriented: `LZ77-index`, `LZend-index`, `RLCSA`, and `SLP` (this one including several improvements upon its previous version [6]). They regard the text as a sequence of characters and report any substring matching the pattern.

Two self-indexes, instead, are word-oriented: `WCSA` and `WSLP` (this one created for this paper). Words and *separators* (maximal spaces between words) are mapped to integers, and the indexes represent the integer sequence. We use the spaceless words model [17], where the single whitespace separator is omitted and assumed by default.

6. EXPERIMENTAL RESULTS

We tested the non-positional and positional scenarios. We used the 108.5 GB Wikipedia collection described by He et al. [12], which is a 10% of the complete English Wikipedia. This collection is formed by 240,179 articles, each of which has a number of versions. There are 8,467,927 versions, of average size 13,757 bytes. We chose a random subset of the articles, and collected all their versions. Each version is considered as a document in our collection. For the non-positional setting our subset contained 24.77 GB, whereas for positional indexes we chose 1.94 GB of random articles.

We consider four query sets, with 1000 queries per set. Two sets are one-word searches, chosen at random from the vocabulary of the indexed subcollection. In the first we take words with up to 1000 occurrences in the subcollection. In the second we take words with more than 1000 occurrences. The other two query sets correspond to phrases of 2 and 5 words chosen at random from the text of the subcollection. For non-positional indexes this is taken as an AND query, whereas for the positional ones it is taken as a phrase query.

6.1 Non-positional indexes

The machine used in this scenario has an Intel(R) Xeon(R) E5520 CPU running at 2.27 GHz, 8 MB cache, 72 GB of RAM memory, 4 cores. The operating system installed is an Ubuntu GNU/Linux version 9.10 running kernel 2.6.31-19-server (64 bits) and `g++` compiler version 4.4.1. Our code was compiled with the `-O9` directive. We measure user times.

We use in these indexes exactly the same parsing of words of He et al. [12], which involves case folding, removing 20 very common stopwords, and no stemming. This reduces the original 108.5 GB to about 85.55 GB, and the 24.79 GB of text we index to about 19.54 GB. Yet we report the space results with respect to the original text size.

We compared a number of variants. The first are representatives of the best classical techniques we are aware of, see Section 2. These include various encodings of the d-gaps with no sampling (`Rice`, `Simple9`, `PforDelta`, and `Vbyte`), and a Vbyte encoding of the d-gaps with samplings in the type of Culpepper and Moffat [8] with $k = 32$ (`Vbyte-CM`), or Sanders and Transier [22] with $B = 128$ (`Vbyte-ST`).

The second group is formed by methods proposed in this paper to handle repetitive sequences. None uses sampling, so intersections are sequential: `Rice-Runs` (Section 3.1), `Vbyte-LZMA` (Section 3.2), `RePair` and `RePair-Skip` (Section 4).

Figure 1 shows the space/time tradeoffs for all the non-positional indexes. The space is given as a percentage of that used by the text in plain form (we are not considering in this experiment the compressed representation of the text itself). The time is shown in microseconds per occurrence.

We can see that, among classical compression methods, the newer methods `Simple9` and `PforDelta` are much better in space than older techniques like `Rice` (one third the space) and `Vbyte` (one fifth the space). In typical collections `Rice` achieves the best space, but these newer methods take advantage of the many runs of 1s. They are also several times faster than `Rice`, and roughly as fast as `Vbyte` on word queries. On conjunctive queries, however, `Vbyte` is faster. In those queries adding samples [8, 22] is advantageous: `Vbyte-CM` and `Vbyte-ST` are significantly faster than `Vbyte` (more than 3 times faster on 5-word queries). The little impact (and even reduction) in size owes to the fact that values stored in the samples are removed from the differential sequence. All the other methods (except `Vbyte-LZMA`) can be combined with such samplings to speed up long queries. Yet, the most important conclusion with regard to classical encoding methods is that they are unsuitable for highly repetitive collections. Our new techniques take one order of magnitude less space, yet they are also significantly slower than the fastest classical variants.

Our first simple method to take advantage of repetitiveness, `Rice-Runs`, makes an important leap in space, from 1%

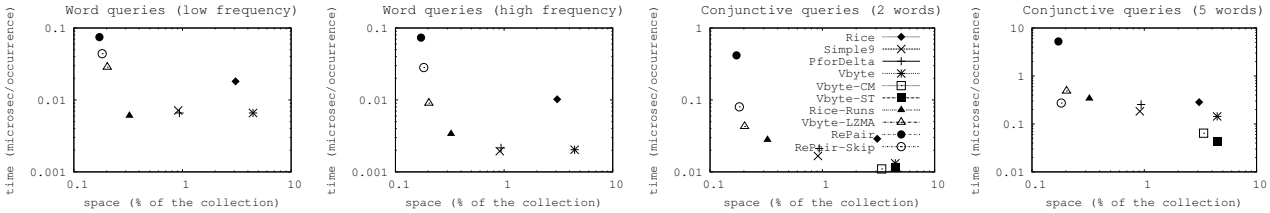

Figure 1: Space/time tradeoffs for non-positional indexes. Logscale.

taken by `Simple9` and `PforDelta` to around 0.3%. It takes one tenth the space of plain `Rice` and is at the same time faster, as it needs much less decompression work. It does not, however, get close to the space of stronger methods like `Vbyte-LZMA`, which achieves around 0.2% by exploiting other types of redundancy, but it is significantly faster than `Vbyte-LZMA` (up to 3 times). We recall that runs can only be exploited if we can arrange the versions consecutively.

`Vbyte-LZMA` is close to the smallest space that we can achieve. It is significantly faster than `RePair` (up to 10 times) and than `RePair-Skip` (up to 3 times) at word queries, as it decompresses faster the inverted list. However, on conjunctive queries, where many of the decoded values have to be discarded, the ability of `RePair-Skip` to skip nonterminals without decompressing them finally makes it almost twice as fast as `Vbyte-LZMA`, and even faster than `Rice-Runs`.

Note that `RePair` obtains lower space (85%) than `Vbyte-LZMA`, despite the fact that LZ77 compression is more powerful than Re-Pair. This is a consequence of LZMA exploiting only intra-list regularities, and shows that significant further repetitions are captured when considering the inter-list redundancies. The skipping information added to Re-Pair adds very little space (6%), but significantly improves time performance (almost 20 times faster on long phrases). This improvement occurs even on one-word queries (up to 2.6 times faster), since `RePair-Skip` does not need to carry out *rank* operations on R_B (recall Section 2.2).

Comparison with previous work. As described in Section 2.1, the best previous work for repetitive collections is by He et al. [12]. Our experiments show that they still obtain roughly half the space and time than `RePair-Skip`. The advantage of `RePair-Skip` is that it works for more general scenarios their techniques are not designed for.

6.2 Positional indexes

For this scenario we used an Intel(R) Xeon(R) E5335 CPU running at 2.00 GHz, 4 MB cache, 16 GB of RAM memory, 8 cores. The operating system installed is an Ubuntu GNU/Linux version 8.04.4 LTS running kernel 2.6.24-29-server (64 bits) and `g++` compiler version 4.2.4. Our code was compiled with the `-O9` directive. We measure user times.

Since self-indexes must reproduce the precise text, we cannot apply case folding nor any kind of filtering in this scenario. We index the original text as is. As explained, word-based self-indexes will regard (and index) the text as a sequence of words and separators. For fairness, the positional inverted indexes will index separators as valid words as well, and phrase queries will choose sequences of tokens, be they words or separators. Still, we note that character-based self-indexes will return more occurrences than word-based self-indexes (or than inverted indexes), as they report all the non-word-aligned ones too. Times per occurrence still seem comparable, yet they slightly favor character-based self-indexes since the time per occurrence drops as more occurrences are reported (there is a fixed time cost per query).

We consider the same techniques of the non-positional setting, now operating on position lists. We used sampling $k = 32$ for `Vbyte-CM` and $B = 64$ for `Vbyte-ST`. We had to adapt `Simple9` because it is unable to represent gaps longer than 2^{28}. While such gaps do not arise on document lists, they occur in position lists. We use the gap $2^{28} - 1$ as an escape code and then the next 32 bits represent the real gap. We exclude `PforDelta` because it has the same limitation, fixing it is more cumbersome, and its performance is not very different from that of `Simple9`. We also exclude `Rice-Runs`, as runs do not arise in the positional setting.

In order to compare fairly with self-indexes, we compress the text with Re-Pair without any sampling and add the space to that of the inverted indexes. This adds just 1.21% of the text size. Adding sampling would slightly increase this space and provide progressively faster extraction time.

We compare the self-indexes described in Section 5: `RLCSA` and `WCSA` (the curves come from using different sampling steps for internal data structures), `SLP` and `WSLP`, and `LZ77-index` and `LZend-index` (the latter using their minimum-space variant [14]). Figure 2 shows the space/time tradeoffs achieved for the four types of queries.

All classical inverted indexes achieve similar space, slightly over 35% ratio. `Simple9` is slightly faster than `Vbyte` for decompressing (i.e., one-word queries), yet `Vbyte` becomes much faster on phrases. Adding sampling, particularly `Vbyte-ST`, improves phrase query times significantly while almost not affecting the space. `Rice` is not competitive.

`RePair` and `RePair-Skip` achieve almost the same space, slightly below 30%, and the latter is always faster for the same reasons as on non-positional indexes. While for words `RePair-Skip` is slower than the classical methods, their times become similar on phrases.

The best space of inverted indexes is achieved by `Vbyte-LZMA`, which reaches a compression ratio near 10%. This represents a significant improvement upon the state of the art. Moreover, for single-word queries its times are only slightly worse than those of `RePair-Skip`, yet on phrase queries its need to fully decompress the list makes it clearly slower, and closer to the times of `Rice`.

Self-indexes are able to use much less space. `WSLP` is only slightly better than `SLP`. This shows that grammar-based compressors do not gain much from handling words instead of characters. They achieve 2%–3% compression ratio. This important reduction in space is paid with a sharp increase in search times. On words they are up to 500 times slower than `Vbyte-LZMA`. This gap, however, reduces to 50 times on 2-word queries and to 5 on 5-word queries. Self-indexes are usually mostly insensitive to the number of words in the

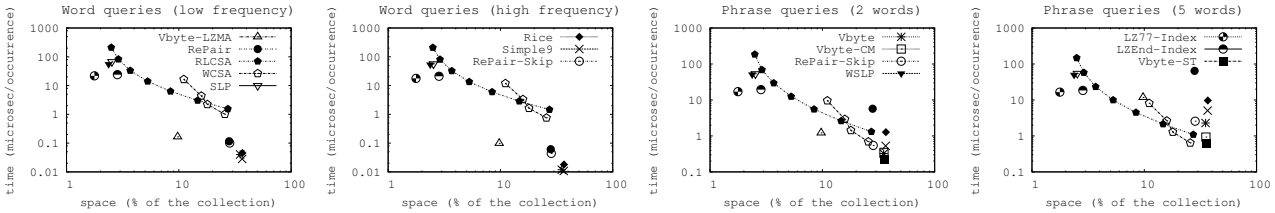

Figure 2: Space/time tradeoffs for positional indexes. Logscale.

query, whereas inverted indexes become much slower when looking for longer phrases.

The `RLCSA` offers a space/time tradeoff that goes from the space of `SLP` (where the latter is faster) to that of classical inverted indexes. These are faster for word searches, but slower as the search phrase becomes longer. The `WCSA` can be regarded as a word-based variant of the `RLCSA`, yet it is not so well optimized for highly repetitive sequences. It is indeed faster than `RLCSA` when using sufficient space. For that space, other indexes are much faster on word queries, but the `WCSA` retains a niche on phrase queries.

Finally, the `LZ77-index` achieves the least space, overcoming its variant `LZend-index` and grammar-based compressors both in time and space. The `LZ77-index` takes less than 2% space and answers queries in less than 20 microseconds per occurrence. Other indexes need at least twice its space to achieve faster query processing.

Acknowledgments

We thank J. He, J. Zeng, and T. Suel, for providing their input collection and help to use it.

This work was funded in part by NSERC of Canada and a David R. Cheriton Scholarship (FC); by Fondecyt Grant 1-110066, Chile (GN); by the Millennium Institute of Cell Dynamics and Biotechnology (ICDB), Grant ICM P05-001-F, Mideplan, Chile (GN and MMP); by Ministerio de Ciencia e Innovación, Grants TIN2009-14009-C02-02 (MMP), TIN2009-14560-C03-02 (AF), and CDTI CEN-20091048 (AF); and by Xunta de Galicia, Grant 2010/17 (AF).

7. REFERENCES

[1] V. Anh and A. Moffat. Inverted index compression using word-aligned binary codes. *Inf. Retr.*, 8:151–166, 2005.

[2] P. Anick and R. Flynn. Versioning a full-text information retrieval system. In *SIGIR*, pages 98–111, 1992.

[3] J. Barbay, A. López-Ortiz, and T. Lu. Faster adaptive set intersections for text searching. In *WEA*, pages 146–157, 2006.

[4] N. Brisaboa, A. Fariña, G. Navarro, A. Places, and E. Rodríguez. Self-indexing natural language. In *SPIRE*, pages 121–132, 2008.

[5] A. Broder, N. Eiron, M. Fontoura, M. Herscovici, R. Lempel, J. McPherson, R. Qi, and E. Shekita. Indexing shared content in information retrieval systems. In *EDBT*, pages 313–330, 2006.

[6] F. Claude, A. Fariña, M. Martínez-Prieto, and G. Navarro. Compressed q-gram indexing for highly repetitive biological sequences. In *BIBE*, pages 86–91, 2010.

[7] F. Claude and G. Navarro. Self-indexed text compression using straight-line programs. In *MFCS*, pages 235–246, 2009.

[8] J. Culpepper and A. Moffat. Compact set representation for information retrieval. In *SPIRE*, pages 137–148, 2007.

[9] S. Ding, J. Attenberg, and T. Suel. Scalable techniques for document identifier assignment in inverted indexes. In *WWW*, pages 311–320, 2010.

[10] R. González and G. Navarro. Compressed text indexes with fast locate. In *CPM*, pages 216–227, 2007.

[11] J. He, H. Yan, and T. Suel. Compact full-text indexing of versioned document collections. In *CIKM*, pages 415–424, 2009.

[12] J. He, J. Zeng, and T. Suel. Improved index compression techniques for versioned document collections. In *CIKM*, pages 1239–1248, 2010.

[13] S. Kreft and G. Navarro. LZ77-like compression with fast random access. In *DCC*, pages 239–248, 2010.

[14] S. Kreft and G. Navarro. Self-indexing based on LZ77. In *CPM*, pages 41–54, 2011.

[15] J. Larsson and A. Moffat. Off-line dictionary-based compression. *Proc. IEEE*, 88(11):1722–1732, 2000.

[16] V. Mäkinen, G. Navarro, J. Sirén, and N. Välimäki. Storage and retrieval of highly repetitive sequence collections. *J. Comp. Biol.*, 17(3):281–308, 2010.

[17] E. Moura, G. Navarro, N. Ziviani, and R. Baeza-Yates. Fast and flexible word searching on compressed text. *ACM Trans. Inf. Sys.*, 18(2):113–139, 2000.

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

[19] G. Navarro and V. Mäkinen. Compressed full-text indexes. *ACM Comp. Surv.*, 39(1):article 2, 2007.

[20] W. Rytter. Application of Lempel-Ziv factorization to the approximation of grammar-based compression. In *CPM*, pages 20–31, 2002.

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

[22] P. Sanders and F. Transier. Intersection in integer inverted indices. In *ALENEX*, 2007.

[23] H. Williams and J. Zobel. Compressing integers for fast file access. *The Comp. J.*, 42:193–201, 1999.

[24] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Trans. Inf. Theory*, 23(3):337–343, 1977.

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

[26] M. Zukowski, S. Heman, N. Nes, and P. Boncz. Super-scalar RAM-CPU cache compression. In *ICDE*, page 59, 2006.