On the Reproducibility of Experiments of Indexing Repetitive Document Collections

Antonio Fariña*,[a], Miguel A. Martínez-Prieto[b], Francisco Claude[c], Gonzalo Navarro[d], Juan J. Lastra-Díaz**,[e], Nicola Prezza**,[f], Diego Seco**,[g]

[a] *University of A Coruña, Facultade de Informática, CITIC, Spain.*
[b] *DataWeb Research, Department of Computer Science, University of Valladolid, Spain.*
[c] *Escuela de Informática y Telecomunicaciones, Universidad Diego Portales, Chile.*
[d] *Millennium Institute for Foundational Research on Data (IMFD), Department of Computer Science, University of Chile, Chile.*
[e] *Universidad Nacional de Educación a Distancia, Spain.*
[f] *University of Pisa, Italy.*
[g] *Department of Computer Science, Faculty of Engineering, University of Concepción, Chile*

Abstract

This work introduces a companion reproducible paper with the aim of allowing the exact replication of the methods, experiments, and results discussed in a previous work [5]. In that parent paper, we proposed many and varied techniques for compressing indexes which exploit that highly repetitive collections are formed mostly of documents that are near-copies of others. More concretely, we describe a replication framework, called `uiHRDC` *(universal indexes for Highly Repetitive Document Collections)*, that allows our original experimental setup to be easily replicated using various document collections. The corresponding experimentation is carefully explained, providing precise details about the parameters that can be tuned for each indexing solution. Finally, note that we also provide `uiHRDC` as reproducibility package.

Keywords: *repetitive document collections, inverted index, self-index, reproducibility.*

1. Introduction

Scientific publications remain the most common way for disseminating scientific advances. Focusing on Computer Science, a scientific publication: *paper*, i) exposes new algorithms or techniques for addressing an open challenge, and ii) reports experimental numbers to evaluate these new approaches regarding a particular baseline. Thus, empirical evaluation is the stronger evidence of the achievements reported by a paper, and its corresponding setup is the only way that the scientific community has for assessing and (possibly) reusing such research proposals.

The ideal situation arises when a paper exposes enough information to make its research easily reproducible. According to the definition proposed by the Association for Computation Machinery (ACM) [1], an experiment is *reproducible* when an independent group of researchers can obtain the same results using artifacts which they develop independently. A weaker form of reproducibility is replicability. In this case,

*Corresponding author
**Reproducibility reviewer
Email addresses: `antonio.farina@udc.es` (Antonio Fariña), `migumar2@infor.uva.es` (Miguel A. Martínez-Prieto), `fclaude@recoded.cl` (Francisco Claude), `gnavarro@dcc.uchile.cl` (Gonzalo Navarro), `jlastra@invi.uned.es` (Juan J. Lastra-Díaz), `nicola.prezza@gmail.com` (Nicola Prezza), `dseco@udec.cl` (Diego Seco)

the original setup is reused, so an experiment is *replicable* when an independent group of researchers can obtain the same results using the author own artifacts. Finally, the ACM document [1] also defines the concept of repeatability. An experiment is *repeatable* if the group of researchers is able to obtain the same results reusing their original setup (including the same measurement procedures and the same system, under the same operating conditions). Non-repeatable results do not provide solid insights about any scientific question, so they are not suitable for publication. Thus, we can accept that research in Computer Science must be, at least, repeatable.

Although reproducibility is the ultimate goal, getting it is not always easy because it requires that a new experimental setup to be reconstructed from scratch. Implementing new algorithms, data structures, or other computational artifacts can be extremely complex, and tuning them may require setting many parameters which, in turn, can depend on particular system configurations or external tools. Besides, it may be necessary to use programs in the exact versions used originally [26]. As a consequence, reproducibility is time-consuming, error-prone, and sometimes just infeasible [4].

In practice, publishing replicable research[1] would be an important step forward in Computer Science. According to Collberg et al. [8], replicating a computational experiment *only* needs that the source code and test case data have to be available. Although simple, many papers do not meet this precondition. The aforementioned paper [8] provides an interesting study involving 601 papers from conferences and journals. It shows that only 32.3% of the papers provide enough information and resources to build their executables in less than 30 minutes. Note that external dependencies must be accessible to compile these sources. Regarding test case data, Vandewalle et al. [30] report that more papers make data available, but it is mainly due to some of them use standard corpora in their corresponding experimental setups. However, the problem goes beyond. Once compiled, proper parameter settings must be set, and sometimes it is tricky to find this information in the papers. In conclusion, reproducibility research is currently challenging.

The situation is not quite different in **Information Retrieval** (IR), the field more related to the research proposed in our work. IR is a broad area of Computer Science focused primarily on providing the users with easy access to information of their interest [3]. As an empirical discipline, advances in IR research are built on experimental validation of algorithms and techniques [20], but reproducing IR experiments is extremely challenging, even when they are very well documented [13]. A recent Dagstuhl Seminar about *reproducibility of data-oriented experiments in e-Science* [12] concludes that IR systems are complex and depend on many external resources that cannot be encapsulated with the system. Besides, it reports that many data collections are private, and their size could be an obstacle even for obtaining them. Finally, it notes that some experimental assumptions are often hidden, making reproducibility a less achievable goal.

[1] In this paper, we use the term reproducibility to refer experimental setups that can be replicated.

Following the initiative of some previous papers [33, 18], the aim of this work is to propose a detailed experimental setup that replicates the methods, experiments, and results on *indexing repetitive document collections* discussed in a previous work [5] (we will refer it as the *parent paper*). This experimental setup focuses on two dimensions: i) the *space* used to preserve and manage the document collection, and the ii) *time* needed to query this data.

The rest of the paper is organized as follows. In Section 2, we briefly describe all the inverted-index based variants and the self-indexes evaluated in the parent paper. We also explain the most relevant space/time tradeoffs reported from our experiments. The following sections are devoted to the reproducibility of these experiments. Section 3 details the fundamentals of `uiHRDC`, our replication framework, which comprises *i)* the *source code* of all our indexing approaches, *ii)* some *test data* collections, and *iii)* the *set of scripts* that runs the main tasks of our experimental setup and generates a final report with the main figures of the parent paper. We focus both on discussing the workflow that leads the process of replicating all our experiments, and also in describing the structure of the `uiHRDC` framework. Section 4 describes the Docker[2] container that allows this workflow to be easily reproduced in a tuned environment. Some brief conclusions are exposed in Section 5 to motivate the need of improving research reproducibility in Computer Science. Finally, Appendix A includes the actual compression ratios obtained for each technique (which complements the results shown in the figures throughout the paper), and Appendix B is devoted to explain how some of our self-indexes can be reused for dealing with universal (not document oriented) repetitive collections.

2. Universal Indexes for Highly Repetitive Document Collections

2.1. Background

Indexing highly repetitive collections has gained relevance because huge corpora of versioned documents are increasingly consolidated. *Wikipedia*[3] is the most recognizable example, exposing millions of articles which evolve to provide the best picture of the reality around us. Each Wikipedia article comprises one version per document edition, and most of them are near-duplicates of others. Apart from versioned document collections, other applications perform on versioned data. For instance, the version control system *GitHub*[4]. In this case, a tree of versions is maintained to control changes from millions of software projects which are continuously updated by their developers. Biological databases (where many DNA or protein sequences from organisms of the same or related species are maintained), or periodic technical publications (where the same data, with small updates, are published over and over) are other examples of computer systems managing highly repetitive data.

[2] `https://www.docker.com/`
[3] `https://www.wikipedia.org/`
[4] `https://www.github.com/`

The parent paper [5] focused on natural language text collections, which can be decomposed into words, and queried for words or phrases. Managing these document collections is a challenge by itself due to their large volume, but at the same time, they are highly compressible due to their repetitiveness. In addition, version control systems require direct access to individual versions. This is also challenging because it demands efficient and potentially large indexes to be deployed on top of the data collection. Thus, we need to compress not only the data, but also indexes required to speed up searches.

The **Inverted Index** [3] has been traditionally used to index natural language text collections. The inverted index maintains a list of the occurrences (also referred as *posting list* or *inverted list*) of each distinct word in the text. It enables two different types of inverted indexes to be distinguished: i) *non-positional* indexes store the list of document identifiers that contain each different word; and ii) *positional* indexes store, in addition to the document identifiers, the word offsets of the occurrences within each document. Posting lists are usually sorted by increasing document identifier, and by increasing word offset within each document for positional indexes. This decision is useful for list *intersections*, a fundamental task under the Google-like policy of treating bag-of-word queries as ranked AND-queries. Intersections are also relevant to solve queries involving multiple words (phrase queries).

There is a burst of recent activity in exploiting repetitiveness at the indexing structures, in order to provide fast searches in the collection within little space. These approaches are summarized in the following. On the one hand, Section 2.1.1 discusses the fundamentals of inverted index compression, and summarizes all our approaches. On the other hand, Section 2.1.2 introduces the concept of self-index [23], an innovative succinct data structure which integrates data and index into a single compressed representation. Besides, we explain how self-indexes can be adapted to perform on document versioned collections attending to our original work. A detailed review of related literature, and a full explanation of our compressed inverted indexes and self-indexes can be found in the parent paper.

2.1.1. Compressed Inverted Indexes

Since posting lists are sequences of increasing numbers, traditional compression techniques typically compute the difference between consecutive values and then encode those *gaps* with a variable-length encoding that favors small numbers. This is the basis of techniques traditionally used to represent posting lists: those using `Rice` codes or `Vbyte` codes, that assign one codeword to each *gap*, or others such as `Simple9`, that packs several *gaps* within a unique integer, or `PforDelta` which compresses blocks of k *gaps*. Therefore, all aforementioned gap encoding techniques take advantage of the fact that the values within posting lists (document identifiers or position values) are probably very close, and consequently they require few bits for their encoding. We have revisited all the previous techniques and we used them in this repetitive scenario. In addition, we have considered other state-of-the-art posting list representations including: `QMX`, which is a good representative of the last generation of SIMD-based techniques [29], or the recent Elias-Fano technique

from [24] (`EF-opt`) and the implementations from [24] of some well-known representations such as `OPT-PFD` [34], `Interpolative` coding [22], and `varintG8IU` [27].

Unfortunately, traditional techniques are not able to detect the repetitiveness that arises in versioned collections. As one of the major contributions of our parent paper, we proposed some different techniques which exploit the types of repetitiveness that arise in versioned collections. We applied them both for non-positional and positional inverted indexes. They are based on run-length encoding (`Rice-Runs`), grammar-based compression (`RePair`), and Lempel-Ziv compression (`Vbyte-LZMA` and `Vbyte-Lzend`).

Even though the use of compressed posting list representations permits to drastically reduce inverted index size, it also has implications in query time. As we indicated in the parent paper, intersections can be performed by traversing the lists sequentially in a *merge*-type fashion. However, if one of the lists to intersect is much shorter than the other/s it is preferred to provide direct access (using sampling) so that the elements of the smallest list can be searched for within the longer list. This type of intersection is commonly referred to as *Set-vs-Set* (*svs*) in the literature. In practice, the best choice is to sort the lists by length. Then, we take the shortest one as the "candidate" list, and it is iteratively intersected with longer and longer lists, hence shortening the candidate list at each step.

There are two main sampling structures to provide direct access in the literature. Culpepper and Moffat [9] propose the first one, referred to as `CM`. It regularly samples the compressed list and stores separately an array of samples, which is searched with exponential search. Given a sampling parameter k, a list of length ℓ is sampled every $k \log_2 \ell$ positions. The second method, by Transier and Sanders [28] (`ST`), applies domain sampling. It regularly samples the universe of positions, so that the exponential search can be avoided. Given a parameter B, it samples the universe of size u at intervals $2^{\lceil \log_2(uB/\ell) \rceil}$. To perform intersections, *lookup* algorithm was defined [28]. It somehow resembles a *merge*-type algorithm but takes advantage of the domain sampling. As we will see below, we combined both `CM` and `ST` sampling with posting list representations based on `Vbyte` and with our Re-Pair-based alternatives.

Posting List Representations. We include here a brief description of the different posting list representations evaluated in our original paper and the tuning options they support (if any). All this information is summarized in Table 1.

- `Vbyte`. We included a simple posting list representation based on `Vbyte` [31] which uses no sampling and consequently performs intersections in a *merge*-wise fashion. We also included two alternatives using `Vbyte` coupled with sampling [9] (called `Vbyte-CM`), with $k = \{4, 32\}$, or domain sampling [28] (called `Vbyte-ST`), with $B = \{16, 128\}$. In the former case, as indicated above, intersections follow a *Set-vs-Set approach* where the smallest list is decompressed and its values are searched for within the other lists using exponential search. For `Vbyte-ST` we used *lookup* intersection algorithm [28]. In addition, we included a hybrid variant of `Vbyte-CM` that uses bitmaps to represent the longest posting

Method	Variants	Description
`Vbyte`	`Vbyte`[31]	Simple Vbyte encoding with no sampling. Intersections are performed in a *merge-wise* fashion.
	`VbyteB`	`Vbyte` enhanced with bitmaps to represent the longest posting lists.
	`Vbyte-CM`[9]	`Vbyte` coupled with list sampling: $k = \{4, 32\}$. Intersections are perfomed in a *set-vs-set* approach.
	`Vbyte-CMB`[9]	`Vbyte-CM` enhanced with bitmaps to represent the longest posting lists.
	`Vbyte-ST`[28]	`Vbyte` coupled with domain sampling: $B = \{16, 128\}$ ($B = \{64\}$ is also tested for the positional scenario). Intersections follow a *lookup* approach.
	`Vbyte-STB`	`Vbyte-ST` enhanced with bitmaps to represent the longest posting lists.
`Rice`	`Rice`[32]	Simple Rice encoding with no sampling. Intersections are performed using a *merge* algorithm.
	`RiceB`	`Rice` enhanced with bitmaps to represent the longest posting lists.
`Simple9` [2]	No variants	Word-aligned `Simple9` that packs consecutive *gaps* into a 32-bit words. Intersections are performed using a *merge* algorithm.
`PforDelta` [15, 35]	No variants	Extends `Simple9` *gaps* to pack many *gaps* together (up to 128). A variant of `Simple9` is used to encode exceptions. Intersections are performed in a *merge-wise* fashion.
`QMX` [29, 19]	No variants	Exploits SIMD-instructions to boost the decoding of long lists. Intersections are performed using an specific algorithm that also benefits from such instructions.
`Rice-Runs` [32]	No variants	`Rice` coupled with *run-length* encoding. Intersections are performed in a *merge-wise* fashion.
`Vbyte-LZMA` [31]	No variants	Encodes *gaps* with `Vbyte` and, if the size of the resulting `Vbyte`-sequence is ≥ 10 bytes, then it is further compressed with *LZMA*. A bitmap indicates which posting lists are compressed with LZMA. Intersections are performed using *merge* algorithm.
`Vbyte-Lzend` [31, 16]	No variants	Encodes *gaps* with `Vbyte` and then all posting lists are compressed as a whole with *LZ-End*. It can be parameterized: $ds = \{4, 16, 64, 256\}$. Intersections first obtain `Vbyte`-encoded sequences and then perfom in a *merge-wise* fashion.
`RePair`-based [17]	`RePair`	Compresses posting lists as a whole using RePair. Intersections are performed using *merge* algorithm over the compressed lists.
	`RePair-Skip`	Adds *skipping* data to `RePair` to improve performance.
	`RePair-Skip-CM`	`RePair-Skip` coupled with `CM`-type (list) sampling: $k = \{1, 64\}$.
	`RePair-Skip-ST`	`RePair-Skip` coupled with `ST`-type (domain) sampling: $B = \{1024\}$ for the non-positional scenario; $B = \{256\}$ for the positional scenario.
Indexes from Ottaviano and Venturini's Framework [24]	`EF-opt` [24]	Partitioned Elias-Fano index.
	`OPT-PFD` [34]	Optimized `PforDelta` variant.
	`Interpolative` [22]	Binary interpolative coding.
	`varintG8IU` [27]	SIMD-optimized Variable Byte code.

Table 1: Summary of posting list representations evaluated in the parent paper.

lists (`Vbyte-CMB`) [9]. In practice, lists longer than $u/lenBitmapDiv$ (where u is the largest document identifier, and $lenBitmapDiv = 8$) are replaced by a bitmap [9] that marks which documents are present in the list. For completeness, we used the hybrid approach over `Vbyte-ST` to build `Vbyte-STB`, and also included a variant `VbyteB` with no sampling.

- `Rice`. We included a representation based on `Rice` codes [32] and also implemented for completeness a `RiceB` variant using bitmaps for the longest lists. In both cases they use no sampling and intersections are done using *merge* algorithm.

- `Simple9`. We included a representation based on the word-aligned `Simple9` technique, by Anh and Moffat [2]. It packs consecutive *gaps* into a 32-bit word. It uses the first 4 bits to signal the type of packing done, depending on how many bits the next *gaps* need: we can pack 28 1-bit numbers, or 14 2-bit numbers, and so on. Intersections are done using *merge* algorithm.

- `PforDelta`. This representation [15, 35] uses the same idea of packing many *gaps* together, typically up to 128 (parameter $pfdThreshold = 128$), while allowing for 10% of exceptions that need more bits than the core 90% of the *gaps*. Those exceptions are then coded separately using a variant of `Simple9`. In our case, we obtained our best results with $pfdThreshold = 100$. Again, we performed intersections *merge*-wise.

- `QMX`. This technique [29] uses SIMD-instructions to boost decoding of large lists,[5] and was coupled with an intersection algorithm [19] that also benefits from SIMD-instructions.[6]

- `Rice-Runs`. We coupled *run-length* encoding with `Rice` coding to boost both the compression and intersection speed of our `Rice` representation in a repetitive scenario where large sequences of $+1$ *gaps* could occur. Basically, a sequence of $+1$ *gaps* of length k is represented as the *ricecode(+1)* followed by *ricecode(k)*. Intersections are performed *merge*-wise, yet they take advantage of the fact that a run of $+1$ values can be skipped/decoded in a unique operation.

- `Vbyte-LZMA`. In this representation we independently compress each posting list with a `Vbyte`+*LZMA* chain. We initially encode the *gaps* in the posting list with `Vbyte` and then compress the output with the *LZMA* variant of LZ77 (`www.7-zip.org`). We use a threshold parameter ($minbcssize$) so that LZMA is only applied on those lists where their `Vbyte` encoded sequence occupies at least 10 bytes ($minbcssize \leftarrow 10$). Otherwise, the initial `Vbyte` encoded sequence is stored. A bitmap indicates which posting lists were stored compressed with `Vbyte` plus LZMA, and which ones only with `Vbyte`. Since `Vbyte-LZMA` only supports extracting a list from the beginning, intersections can involve a initial

[5] `http://www.cs.otago.ac.nz/homepages/andrew/papers/QMX.zip`.
[6] `https://github.com/lemire/SIMDCompressionAndIntersection`.

step that applies LZMA decoder to recover the `Vbyte` encoded sequences, and then we apply *merge* algorithm on those `Vbyte` encoded sequences.

- `Vbyte-Lzend`. This representation follows the same idea of `Vbyte-LZMA` but uses LZ-End [16] instead of LZMA to compress the posting lists. Since LZ-End allows random access, the sequence of all the concatenated lists can be compressed as a whole, not list-wise. Therefore, we first concatenate the `Vbyte` representation of all the posting lists (keeping track of the offset where each posting list started in the `Vbyte` encoded sequence), and then use LZ-End to represent it. At construction time, we can obtain different space/time tradeoffs by tuning the *delta-codes-sampling* parameter (ds) of LZ-End (see [16] for details). In our experiments we set it to $ds = \{4, 16, 64, 256\}$. At intersection time, we first recover the `Vbyte` encoded sequences and then proceed *merge*-wise as in `Vbyte-LZMA`.

- `RePair`-based representations. As in the LZ-End-based representation we compress all the posting lists as a whole, yet we directly work on the sequences of *gaps* rather than on their `Vbyte` encodings. We use the grammar-based compressor Re-Pair that recursively replaces the most frequent pair of symbols (initially called *terminals*, which are those *gaps* from the concatenation of all the posting lists) by a new *non-terminal* symbol not occurring before in the original sequence. Re-pairing process ends when no pair occurs at least twice. The result is a sequence that could contain both terminal and non-terminal symbols, and a dictionary of substitution rules which is represented in a compact format. This makes up our basic `RePair` representation. It allows us to extract individual posting lists,[7] and to perform intersections in a *merge*-type fashion over the compressed representation of the lists (non-terminals are expanded/decoded recursively during the traversal of the lists). In addition, in our implementation we added a parameter $repairBreak$ that allows us to stop the recursive pairing when the gain in compression ratio (reduction of size of the compressed sequence) of the current step with respect to the previous step is smaller than $repairBreak$.[8] In practice, for `RePair` we set $repairBreak = 0.0$ so that the Re-pairing process is not broken at all.

  The compact representation of the dictionary of rules allows us to add additional *skipping* data (i.e. the sum of all the *gaps* included below that non-terminal symbol) with little space cost. This skipping data allows us to improve both decompression time for a list, and intersection time. We adapted the *merge*-based algorithm from `RePair` so that it is boosted by using this skipping data. The resulting algorithm was named *skip*. This makes a new representation of posting lists referred to as `RePair-Skip`.

[7] To avoid creating Re-Pair phrases that span along two different posting lists, we add a non-repeating separator to mark the beginning of each posting list.

[8] Being n and m respectively the length of the original sequence and the length of the compressed sequence, we initially set $prevRatio \leftarrow (100.0 * m/n)$. Then, after each substitution, if $((prevRatio - (100.0 * m/n)) < repairBreak)$ we break the Re-pairing process. Otherwise, we simply update $prevRatio \leftarrow (100.0 * m/n)$.

In our experiments, we tuned `RePair-Skip` with $repairBreak = 4 \times 10^{-7}$ and $repairBreak = 5 \times 10^{-7}$ respectively for the non-positional and positional scenarios.

Initially, neither `RePair` nor `RePair-Skip` had sampling to speed up intersections, and only `RePair-Skip` could benefit of the additional data to boost the intersections. However, since `RePair-Skip` was shown to outperform `RePair`, we also created two `RePair-Skip` variants using both `CM`-type and `ST`-type sampling. They are named `RePair-Skip-CM` (where we set $k = \{1, 64\}$) and `RePair-Skip-ST` (with $B = \{256, 1024\}$).

- `Ottaviano&Venturini's Partitioned Elias-Fano indexes`. In [24], Ottaviano and Venturini presented several posting list representation alternatives based on Elias-Fano codes. We used the best of them (`EF-opt`). The source code is available at authors' website.[9] In addition, they included in their framework implementations of some well-known representations such as `OPT-PFD` (optimized `PforDelta` variant [34]), `Interpolative` (Binary Interpolative Coding [22]), and `varintG8IU` (SIMD-optimized Variable Byte code [27]). All of them were compared in our parent paper.

Our implementation of non-positional and positional inverted indexes. Given a text T that is composed of a set of documents, to create our inverted indexes, we process T to gather the different words/terms in T, and for each term we obtain the positions were they occur. The text itself can be processed with a suitable compressor in order to reduce space needs. In our implementation, we used Re-Pair[10] as text compressor, which reached compression ratios below 2% (we kept the rules uncompressed -as a pair of integers- to speed up decompression), and we added sampling for every $\{1, 2, 8, 32, 64, 256, 4096\}$ symbols of the final Re-Pair sequence to allow decompressing arbitrary parts of the original text.[11] Yet, the main focus of our parent paper was set on how to efficiently deal with the set of posting lists. We included all the above posting lists representations, with the exception of Ottaviano&Venturini's implementations, within a package that provides us methods to build a compressed representation for a set of posting lists, extract a list, intersect 2-n lists, show the compressed size, etc. We did this to obtain compressed representations for posting lists to be used in a non-positional scenario, where we indexed document-ids and intersections were used to solve AND-conjunctive queries for the elements of our query patterns, but also for a positional scenario, where those patterns were taken as phrase queries. In this positional scenario, we indexed actual word/term positions and solving phrase queries implied that the intersections had to consider the order of the terms in the query patterns to return the documents where they occur at consecutive positions. The building process is handled by a **build** program that processes the source text and saves all the required structures to disk.

[9] `https://github.com/ot/partitioned_elias_fano`.

[10] We have also the option to use: a LZ-End compressor; keeping the text in plain form; or even not storing the text at all.

[11] This was of interest in the positional scenario where, as we will see in the next section, we are interested in comparing the snippet extraction time with that of self-indexes.

A **searcher** program is on charge of loading those structures into main memory and then allowing us to perform queries. Both programs are linked a posting list representation and a text compressor.

According to the experiments reported in our parent paper, our compressed inverted index variants allow us to perform two types of queries:

- `locate(p)` returns the positions of all the occurrences of p in T. In the non-positional scenario our experiments (see Section 2.2) were focused on comparing both the fetch time and the intersection time of the different posting lists representations, so basically `locate(p)` returns the documents where p occurs.

  In the positional scenario, `locate(p)` returns a pair of the form (doc, pos_in_doc). Note that our posting list representation indexes *absolute* positions in T rather than pairs (doc, pos_in_doc). Therefore, to support returning relative positions within each document, we enhanced our inverted index with an array of offsets that mark the beginning of each document in T, and provided an operation `merge-occs-to-docs` which efficiently maps a sequence of absolute positions into pairs (doc, pos_in_doc). Therefore, to solve `locate(p)` we initially perform an intersection to gather the positions in T where p occurs, and then perform `merge-occs-to-docs` operation to obtain the final output.

- `extract(a,b)` retrieves the text fragment in $T[a, b]$. This is efficiently supported by using the regular sampling added to the (Re-Pair) text representation.

2.1.2. Self-Indexes

A *self-index* [23] is a compact data structure that enables efficient searches over an string collection (called the text, T), and also replaces the text by supporting extraction of arbitrary snippets. Thus, self-indexes provide, at least, two basic queries:

- `locate(p)` returns the positions of all the occurrences of p in T.

- `extract(a,b)` retrieves the text fragment in $T[a, b]$.

This functionality allows self-indexes to be considered as an alternative choice to positional inverted indexes.

Our original setup comprised three families of self-indexes that were able to capture high repetitiveness. All of them have a representative proposal which regards T as a sequence of characters, but two word-oriented approaches (which process T as a sequence of words) were also proposed. It is worth noting that, in all cases, `locate(p)` returns *absolute* positions in T that must be then converted into (document,offset) pairs. The aforementioned `merge-occs-to-docs` position-document mapping is used again for such purpose.

CSA-based Self-Indexes. The Compressed Suffix Array (CSA) by Sadakane [25] is one of the pioneering self-indexes and proposes a succinct suffix array (SA) encoding. It retains the original locate functionality of the suffix array (based on binary search), while providing snippet extraction in compressed space. CSA encompasses two main structures: a bitmap B, which marks where the first symbol of the suffixes changes in SA, and Ψ, an array which stores the position in SA pointing to the next character of a suffix. Ψ is highly compressible, but it is also massively used to decode a portion of the text. Thus, self-indexes dealing with highly repetitive collections must balance Ψ compression and decoding efficiency to be competitive with respect to compressed positional inverted indexes.

Two different CSA-based approaches were evaluated in our benchmark: `RLCSA` and `WCSA`.

- `RLCSA`. The Run-Length Compressed Suffix Array, by Mäkinen et al. [21], exploits that Ψ contains long runs of successive values in highly repetitive collections. It performs run-length encoding of $\Psi(i) - \Psi(i-1)$ and stores regular samples to allow efficient access to absolute Ψ values. This *sample* value must be provided to build `RLCSA`, and yields different space/time tradeoffs: larger *sample* values are used to reduce space requirements, but it penalizes data access speed; on the contrary, small *sample* values increase efficiency at the price of less compression. The authors proposed *sample*= 512, but our experiments consider 7 different values of the form 2^i, from $i = 5$ (*sample* = 32) to $i = 11$ (*sample* = 2048), to analyze particular tradeoffs. A *blocksize* parameter is also required to configure internal bit vectors blocks (it sets the number of reserved bytes per block). We use *blocksize* = 32, as suggested by the authors.

- `WCSA`. The Word Compressed Suffix Array, by Fariña et al. [11] adapts CSA to cope with particular features of natural language; i.e. it processes the input text as a sequence of words instead of characters. `WCSA` transforms the original text into an integer sequence where each position refers to a word/separator, but it does not provide any particular optimization to manage highly-repetitive texts. We included it in our original experiments because it acts as a bridge between inverted indexes and more sophisticated self-indexes, both in space and time complexity. `WCSA` builds B and Ψ, but over the integer sequence, and provides word-based location and extraction. This requires keeping samples of SA and samples of the inverse of SA (indicating which position of SA points to the j-th word) at regular intervals. The non-sampled values can be recovered with Ψ. Therefore, three different parameters must be provided at construction time: $\langle sPsi, sA, sAinv \rangle$. Note that the inverse of SA is only needed for extract, SA is used both for extract and locate, and Ψ is used in all search operations. Therefore, even though we can tune different setups yielding similar space requirements, it is worth to keep a rather small value of $sPsi$, a larger value of sA, and an even larger value of $sAinv$. We evaluated seven different configurations for the sampling parameters ranging from $\langle sA, sAinv, sPsi \rangle = \langle 8, 8, 8 \rangle$ to $\langle 2048, 2048, 2048 \rangle$. In particular, we used the values:

$\{\langle 8,8,8\rangle, \langle 16,64,16\rangle, \langle 32,64,32\rangle, \langle 64,128,32\rangle, \langle 128,256,128\rangle, \langle 512,512,512\rangle, \langle 2048,2048,2048\rangle\}$.

SLP-based Self-Indexes. We previously motivated that grammar-based compression is a good choice for posting list encoding, but it is also promissory for self-indexes. Our original paper focused on a particular type of grammar compressor built around the notion of straight-line program (SLP).[12] We explored two SLP-based self-indexes, referred to as `SLP`and `WSLP`, which were carefully tuned for our experiments.

- `SLP`. Claude et al. [6] proposed originally a character oriented SLP self-index to manage (highly repetitive) biological databases, and then optimized it to cope with natural language collections [7]. `SLP` indexes the set of rules as a labeled binary relation, and the reduced sequence (obtained by RePair) using a varied configuration of bit-based structures. `SLP` requires space proportional to that of a Re-Pair compression of the text, but its performance is less competitive than that reported by other self-indexes. The original `SLP` implementation was improved in our parent paper, where we tunned some of its algorithms and internal data structures. A single q parameter is required to build an SLP index; it sets the lengths of the q-grams that are indexed, by an internal dictionary, to improve prefix/suffix searches. q is set by default to 4 characters because no relevant improvements have been found for larger values.

- `WSLP`. The word-oriented SLP was proposed in our parent paper, and adapted `SLP` to perform on a sequence of integers (word identifiers). Its internal configuration is similar to that of the `SLP`, but it does not use the q-gram dictionary because it was not competitive for words. Thus, `WSLP` exposes a simple build interface which does not require any parameter.

LZ-based Self-Indexes. Finally, we evaluated two different approaches of self-indexing based on two variants of LZ77-like parsing: `LZ77-index` and `LZend-index`. In short, an LZ77-like parsing transforms a text into a sequence of phrases, each one encoding the first occurrence of a substring. Each substring concatenates a maximal substring previously used in the text and a trailing character. `LZ77-index` and `LZend-index` uses a similar configuration of succinct data structures to encode their corresponding structure of phrases, and to allow fast search and decode capabilities.

Both self-indexes were originally proposed by Kreft and Navarro [16], but we tuned them to improve their original space/time tradeoffs. Besides, it is worth noting that they support five different configurations.[13] The *default* configuration (`Conf.#1`) is considered if no parameters are provided. It reports the best performance but also the worst space. The following parameter configurations are also allowed: ``‘‘bsst ssst’’`` (for

[12]SLPs are grammars in which a rule X_i generates i) a terminal j, or ii) a pair of non terminals $X_l X_r$.

[13]These configurations are described in our original paper and basically consider different structures and search algorithms to provide `locate` and `search` functionality.

Conf.#2), ‘‘brev’’ (for Conf.#3), ‘‘bsst brev ssst’’ (for Conf.#2), and ‘‘bsst brev’’ (for Conf.#5). The latter reports larger space savings at the price of a bit slowdown, but ensuring a competitive performance. We chose it in our benchmark.

2.2. Experimental Results

We experimentally studied the space/time tradeoffs obtained when performing `locate` operation with the described posting list representations, in both the non-positional and positional scenarios. In the positional scenario we also added a comparison with the proposed self-indexes, and we finally included results regarding the time needed to `extract` snippets.

In this section we provide a summary of the results that can be obtained using our replication framework `uiHRDC` (see Section 4). In particular, for `locate(p)` operation we include results for the case where the patterns are 2-word phrases, and for `extract(range)` we only include results for snippets of around 13,000 characters. In Section 3.1 we discuss that there are actually four types of patterns for `locate` and two different snippet lengths for `extract` operation. In that section we also discuss details of the text collections used. In addition, we have also included some fixes to errors that were detected in the parent paper during the reviewing process.

The time measures included here are referred to CPU user-times and were obtained using our Docker instance in an Intel(R) i7-8700K@3.70GHz CPU with 64 GB of DDR4@2400MHz memory.

2.2.1. Fixes to the Parent Paper

Before delving into the details of the experiments described in the parent paper, it is worth noting that developing `uiHRDC` helped us to detect some minor errors in the results reported in the parent paper. More precisely:

- `QMX` performance was wrong in the experiment dealing with `locate` operation for 5-words phrase patterns (Figure 3 of the parent paper) in the non-positional scenario. Its actual performance is slightly faster than the one reported by `VbyteB`, but it means that it is half an order of magnitude slower than the fastest choice: `Vbyte-STB`. The `locate` times of `QMX` were also wrong in the case of word queries in the positional scenario (Figure 6 of the parent paper). Although it remains as the fastest choice in both cases, its difference to `Simple9` or `Vbyte` is drastically reduced.

- The `OPT-PFD` compression ratios are wrong in the experiments for the positional scenario (Figures 6 and 9). Its actual compression ratio is 29.742% (instead of 31.848%), so it is more effective than `Interpolative` and `EF-opt`.

2.2.2. Non-Positional scenario: Inverted Indexes

For the non-positional scenario we include a comparison of all the compressed inverted index representations discussed in Section 2.1.1 when considering `locate` operation. Yet, in this case we only consider fetch/intersection time and skip the parsing time of the query patterns. That is, we assume all the patterns have been mapped into *ids* before timing starts.

Figure 1 shows the results. In the left part, we show the results for the state-of-the-art techniques. In the right part we also include our proposals. Recall that for the variants from `Ottaviano&Venturini's` framework [24] we used exactly their source code and simply adapted the format of our data and patterns to run their `build` and `search` programs. Those techniques are marked with an '`*`' in the figures.

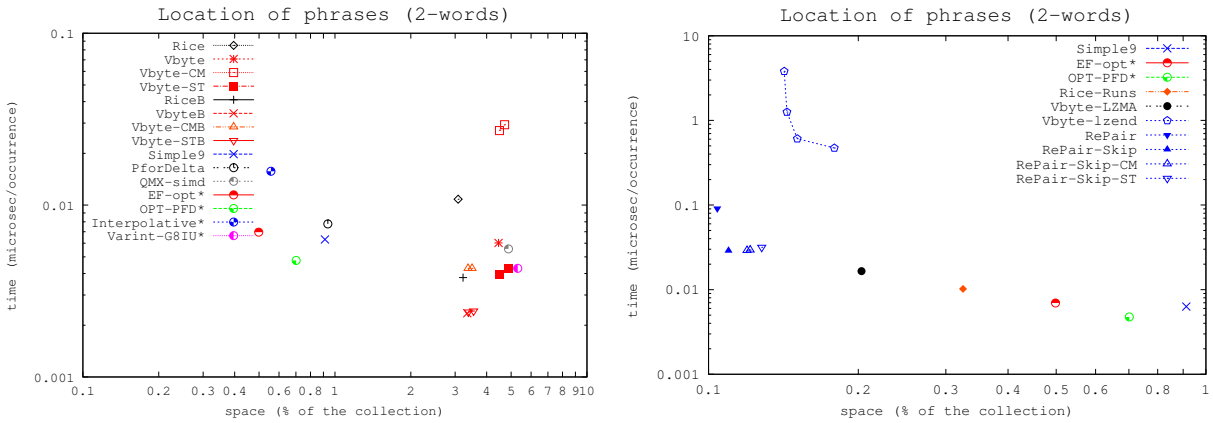

Figure 1: Main results for the non-positional scenario.

As shown in the parent paper, our new techniques `Rice-Runs` `Vbyte-LZMA`, `Vbyte-Lzend`, and Re-Pair-based variants drastically reduced the space needs of the existing techniques. Yet, they were typically also much slower (particularly in the case of `Vbyte-Lzend`).

2.2.3. Positional scenario: Inverted Indexes and Self-Indexes

In this scenario we compare both our inverted index representations and self-indexes at both `locate` and snippet `extract` operations.

Pattern Location. We did not include all the inverted index variants used in the non-positional scenario, but only those that could be of interest here. For the self-indexes, timing includes the time needed to perform `(locate(p))` operation to obtain the positions where `(p)` occurs, and then converting absolute positions to $(document, offset)$ pairs using `merge-occs-to-docs` operation. For the inverted indexes, timing includes: *(a)* the parsing time required to map each pattern into a sequence of *ids*; *(b)* performing intersection of the required posting lists (for those *ids*); and *(c)* running `merge-occs-to-docs` to obtain the final result. For the techniques from `Ottaviano&Venturini's` framework we directly used their sources *with no modifications* to

measure *(b)* times, whereas time measures for stages *(a)* and *(c)* where done separately with a program we also implemented. During step *(a)*, our program not only measures times, but also outputs the sequence of *ids* obtained after the parsing of each pattern `p`. These *id*-based patterns are used to perform intersections with the techniques from `Ottaviano&Venturini's` framework.

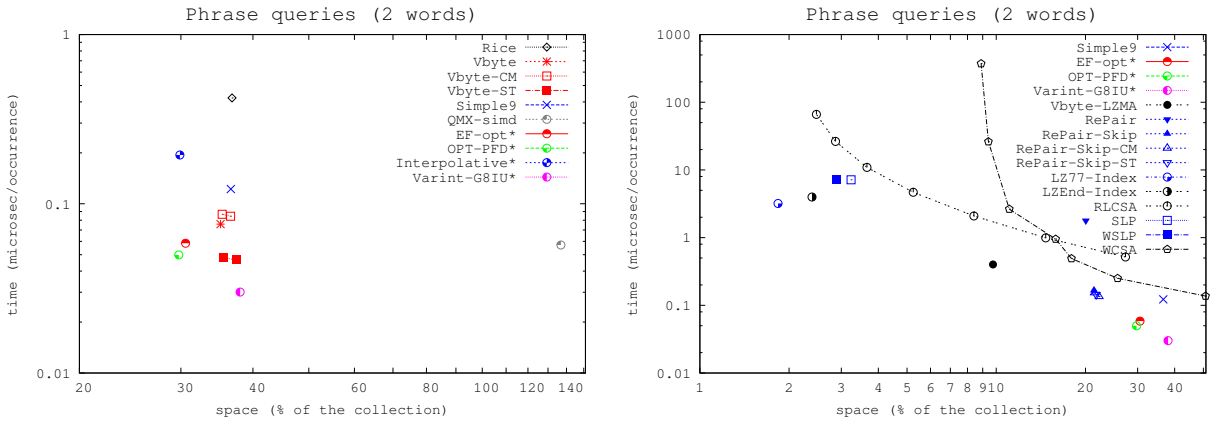

Figure 2: Main results for the positional scenario.

Figure 2 shows the results. On the one hand, our `Vbyte-LZMA` and Re-Pair-based inverted indexes still improve the space needs of the other representations, while being typically up to one order of magnitude slower. On the other hand, self-indexes showed to obtain around one order of magnitude reduction on space, while becoming up to three orders of magnitude slower than the fastest inverted index alternative.

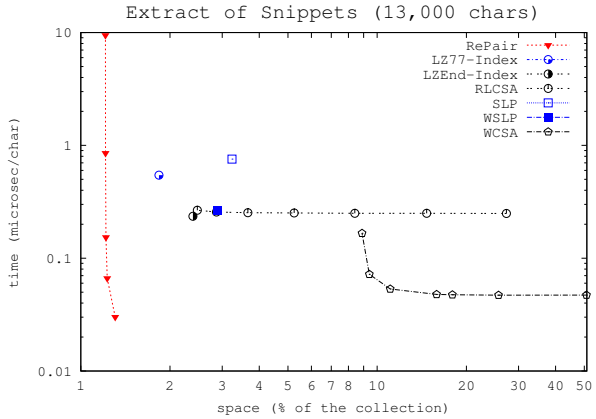

Figure 3: Main results regarding text extraction performance for both self-indexes and text compressed with Re-Pair.

Snippet Extraction. We report the time needed to extract text snippets $T[s, e]$ from the self-indexes and compare them with the decoding performance of Re-Pair, that was the text compressor chosen to compress the document collection for the representations using inverted indexes. Recall that Re-Pair is coupled with

the additional sampling that permit partial decompression, as discussed in Section 2.1.1. Results are shown in Figure 3. We can see that Re-Pair is very fast at decompression (when a dense sampling is used) and requires less than 1.5% of the size of the document collection. With very similar space needs (around $2-3\%$) we find most of the self-indexes (`LZend-index`, `LZ77-index`, `RLCSA SLP`, and `WSLP`), yet they are typically much slower. Finally, `WCSA` competes in speed with Re-Pair, but requires around one order of magnitude more space.

3. The `uiHRDC` Framework

Our experimental framework was named `uiHRDC` *(universal indexes for Highly Repetitive Document Collections)*. It is licensed under the GNU Lesser General Public License v2.1 (GNU LGPL), is hosted at a GitHub repository,[14] and it is also published through Mendeley Data [10]. It includes all the required elements to reproduce the main experiments in our original paper including datasets, query patterns, source code, and scripts.

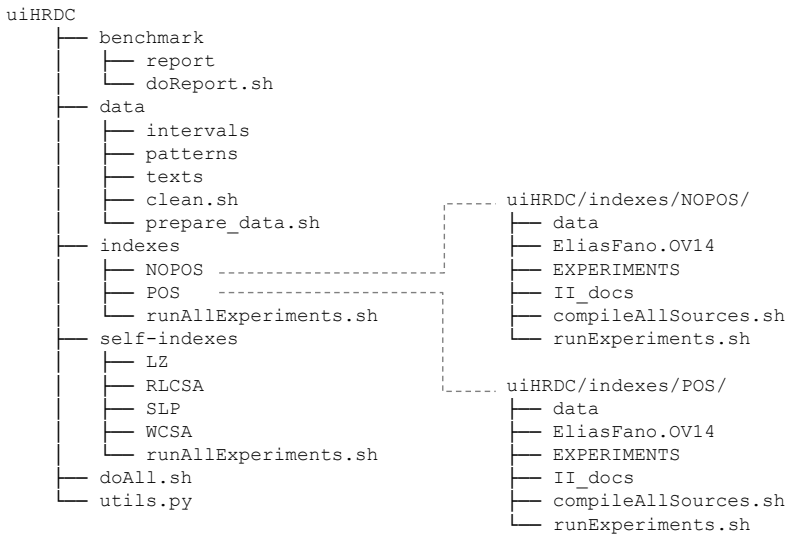

Figure 4: Structure of the `uiHRDC` repository.

The general structure of the `uiHRDC` repository is shown in Figure 4. It can be seen that, under the root of the repository, we include: *i)* directory `benchmark` which includes a LaTeX formatted report and a script that will collect all the data files resulting from running all the experiments and will generate a PDF report with all the relevant figures (including those in Section 2.2). Further python scripts required for such task are also included in directory `utils.py`. *ii)* Directory `data`, which includes the text collections (7z

[14] `https://github.com/migumar2/uiHRDC/`

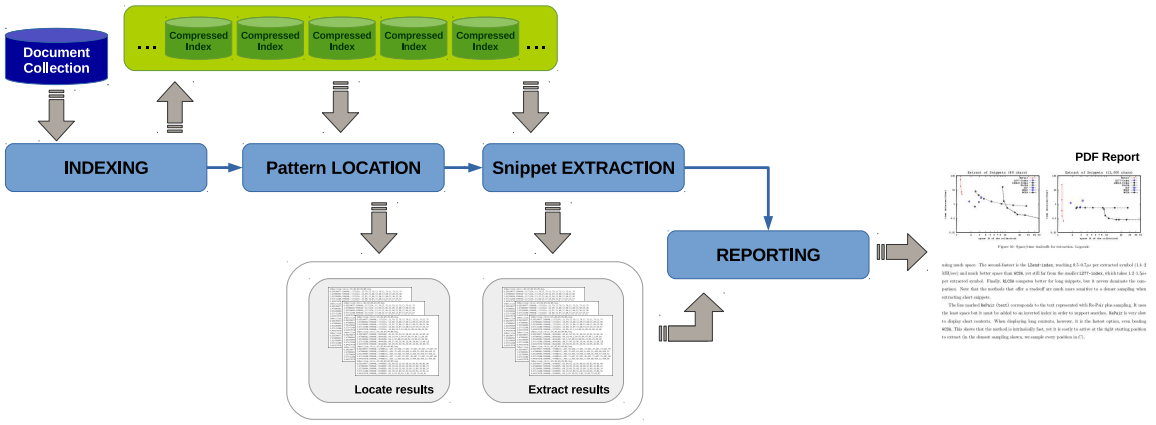

Figure 5: Workflow used in `uiHRDC` to reproduce our experiments.

compressed), and the query patterns. *iii)* Directories `indexes` and `self-indexes` that contain the source code for each indexing alternative, and scripts that permit to run all the experiments for each technique. This includes the construction of each compressed index of interest (using a **builder** program) and then performing both `locate` and `extract` operations over that index (using the corresponding **searcher** program). Each experiment will output relevant data to a results-data file. And *iv)* a script `doAll.sh` that will drive all the process of decompressing the source collections; compiling the sources for each index and running the experiments with it; and finally, generating the final report. The overall workflow followed is depicted in Figure 5.

In the following section we will include further details regarding the contents of our repository.

3.1. Test Data

Within `uiHRDC` we provide in `data` directory, both the document collections and the query sets used in the experimental setup of our parent paper. They are described below.

3.1.1. Document Collections

Our document collections were created from the 108.5 GB Wikipedia collection described by He et al. [14], which contained 10% of the complete English Wikipedia from 2001 to mid 2008. It contains 240,179 articles, and each of them has a number of versions. Its statistics are shown in Table 2. Note that we did not have the original 108.5 GB collection, but a filtered (tag-free) version of it whose size totaled 85.58 GB. Therefore, the original collection was 1.27 times larger than ours.

From the `Full` collection of articles, we chose two subsets of the articles, and collected all the versions of the chosen articles. For the non-positional setting our subset (`Non-pos`) contains a prefix of 19.53 GB of the full collection, whereas for positional indexes we chose 1.94 GB of random articles. However, for a fair

Subset	Size (GB)	Articles	Number of versions	Versions / article	Filename (within `uiHRDC`)	Filesize (GB)
Full	108.50	240,179	8,467,927	35.26	--	85.58
Non-pos	24.77	2,203	881,802	400.27	text20gb.txt	19.53
Pos	1.94	4,327	149,761	34.61	wiki_src2gb.txt	1.94

Table 2: Detailed statistics of the document collections used.

comparison with the techniques from [14], in the non-positional scenario we scaled the size of the `Non-pos` subset using the above 1.27 factor when referring to its space requirements. Additional statistics of our two subsets are also included in Table 2.

3.1.2. Query sets

Since the experiments target at providing space/time for the different indexing alternatives when performing `locate(pattern)` and `extract(interval)` queries we provide two types of query sets for each document subcollection.

- Query sets for `locate`: We provide four query sets, each of them containing 1,000 queries, which include: *i)* two query sets composed of one-word patterns chosen at random from the vocabulary of the indexed subcollection. In the first case (W_a), it includes low-frequency words occurring less than 1,000 times. In the second case, the query set (W_b) includes high-frequency words occurring more than 1,000 times; *and ii)* two query sets with 1,000 phrases composed of 2 and 5 words that were chosen randomly from the text of the subcollection (with no restrictions on its frequency).

- Interval sets for `extract`: Aiming at measuring extraction time when recovering snippets of length 80 (around one line) and 13,000 (around one document, in our collection) characters, we generated: *i)* a set of 10,000 intervals of width 13,000 characters from the `POS` text collection, and *ii)* a set containing 100,000 intervals of width 80 characters. Since these intervals are not suitable for our word-based self-indexes (`WCSA` and `WSLP`), and assuming that the average word length is around 4 in our datasets, we also generated two additional sets composed respectively of 10,000 intervals containing $3,000$ words each, and 100,000 intervals containing 20 words each.

3.2. Source Code and scripts

As indicated above, the source code provided in our `uiHRDC` repository has two main directories `indexes` and `self-indexes`. Those directories include both the source code and the scripts required to reproduce our experiments. In this section we include more details regarding their structure so that an interested reader can rapidly understand how they are organized.

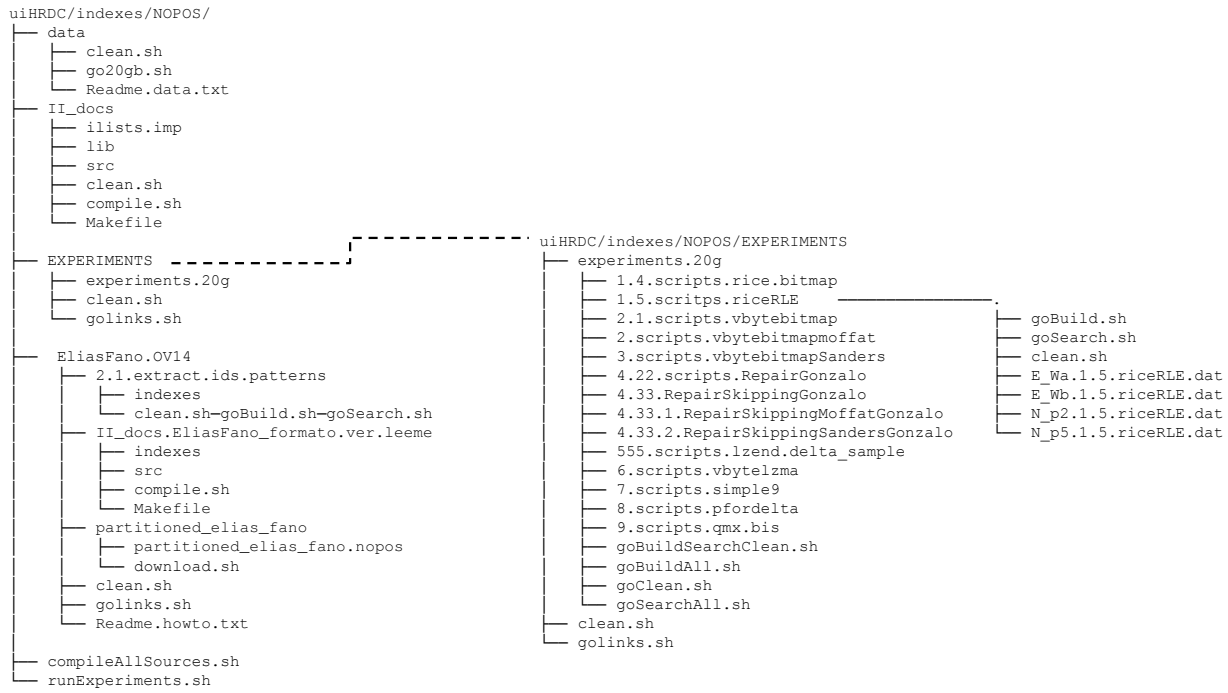

Figure 6: Structure of the `uiHRDC/indexes` directory in the `uiHRDC` repository.

3.2.1. Indexes

Under `uiHRDC/indexes` directory, as it is shown in Figure 4, we can find a script `runAllExperiments.sh` and two directories `NOPOS` and `POS`. Basically, that script enters both directories, and then runs two scripts: one to compile the source codes (`compileAllSources.sh`) and another one to launch the experiments (`runExperiments.sh`) in that directory. An interested reader should probably start by opening those small scripts. These scripts are included in Figure 6, where we show the structure of directory `NOPOS` (the structure of `POS` is almost identical).

Directory `uiHRDC/indexes/NO-POS`: Non-Positional Inverted Indexes. Under this directory, we can find the scripts `compileAllSources.sh` and `runExperiments.sh` discussed above, a `data` directory were links to `uiHRDC/data` will be created by the scripts under this directory, and finally two main parts: *i)* `EliasFano.OV14` with the sources and scripts needed to reproduce our experiments for techniques `EF-opt`, `Interpolative`, `varintG8IU`, and `OPT-PFD`; and *ii)* Directories `II_docs` and `EXPERIMENTS` that include respectively the source code of the remaining inverted index variants and the corresponding scripts to run the experiments. We include more details below.

- Our implementation and scripts within directories `II_docs` and `EXPERIMENTS`. We have organized the source code for our inverted indexes within `II_docs` directory. In Figure 6, we can see a script

`compile.sh` and three subdirectories: `ilists.imp`, `lib`, and `src`. The implementation for all our types of compressed posting list representations is contained within directory `ilists.imp`. Script `compile.sh` will create a package for each of them and will move such package into `lib` directory. Finally, the source code for our non-positional compressed inverted index, located within `src` directory, will be linked with each of those posting list representations to obtain the final `BUILD*` and `SEARCH*` binaries for each all our variants of non-positional inverted index.

In Figure 6, we can also see the contents of directory `EXPERIMENTS/experiments20gb`. Basically, there is a subdirectory for each technique with scripts `goBuild.sh` (to create the indexes), `goSeach.sh` (to perform query operations), and `clean.sh` (to clean temporal stuff). Those techniques are respectively (top-to-bottom in the figure): *1.4)* `Rice` and `RiceB`, *1.5)* `Rice-Runs`, *2.1)* `Vbyte` and `VbyteB`, *2)*`Vbyte-CM` and `Vbyte-CMB`, *3)* `Vbyte-ST` and `Vbyte-STB`, *4.22)* `RePair`, *4.33)* `RePair-Skip`, *4.33.1)* `RePair-Skip-CM`, *4.33.2)* `RePair-Skip-ST`, *555)* `Vbyte-Lzend`, *6)* `Vbyte-LZMA`, *7)* `Simple9`, *8)* `PforDelta`, and *9)* `QMX`. In addition, a script `goBuildSearchClean.sh` is in charge of entering each directory running the experiments for the corresponding technique. The output of those experiments are data files containing both space and time statistics (see files `E_Wa.1.5.riceRLE.dat, ...`).

- Ottaviano&Venturini's variants within directory `EliasFano.OV14`: This directory contains: *i)* A subdirectory `partitioned_elias_fano` containing the source code from Ottaviano&Venturini's framework and a script to run the experiments for variants `EF-opt`, `Interpolative`, `varintG8IU`, and `OPT-PFD`; and *ii)* two additional directories `II_docs.EliasFano_formato.ver.leeme` and `2.1.extract.ids.patterns` that contain source code and scripts to transform the text-based patterns into the *id*-based patterns that will be used at query time. In this case, the output of the experiments is written to a log-file and finally a Phyton script parses such file to gather the values regarding space and time measures for each technique.

Directory ***uiHRDC/indexes/POS****: Positional Inverted Indexes.* As indicated above, the structure of this directory is almost identical to that of directory `uiHRDC/indexes/NO-POS`. Therefore, we include no further details here.

3.2.2. Self-Indexes

Under directory `uiHRDC/self-indexes`, as it is shown in Figure 7, we can find a script `runAllExperiments.sh` and one directory for the different types of self-indexes. In particular, we find directories `RLCSA`, `WCSA`, *LZ* (for `LZ77-index` and `LZend-index`) and `SLP` (for `SLP` and `WSLP`). In those directories we will find, among others, the source code and scripts to run the experiments for each technique.

In addition, directory `uiHRDC/self-indexes/collectResults` contains scripts to process the log files obtained when we run the script `runAllExperiments.sh` and create gnuplot-type data files for each technique

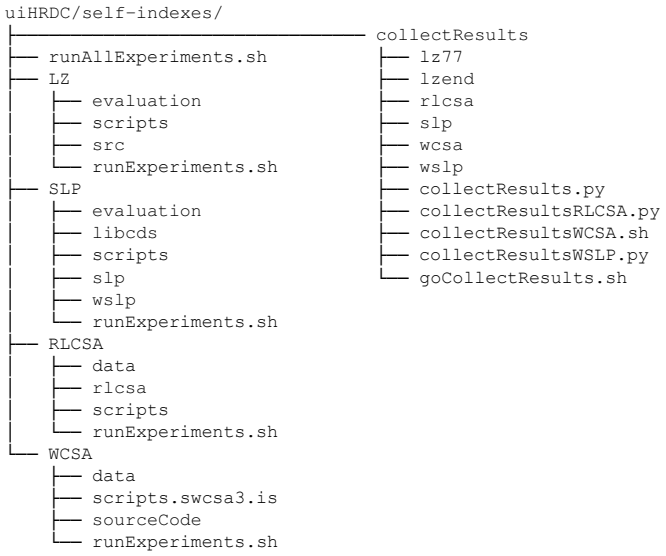

Figure 7: Structure of the `uiHRDC/self-indexes` directory in the `uiHRDC` repository.

that are used to make up the final report for the experiments.

4. Deploying the Experimental Setup with Docker

For those readers interested in reproducing our experiments in their machine, we provide a *Docker* environment that will allow them to:

1. Reproduce our test framework. We create a docker image with Ubuntu 14 (ubuntu:trusty) that includes all the libraries and software requirements to compile/run our indexing alternatives and also build the final report. These includes packages that are installed via `apt` such as: gcc-multilib, g++-multilib, cmake, libboost-all-dev, p7zip-full, openssh-server, screen, texlive-latex-base, and texlive-fonts-recommended; and finally, snappy-1.1.1[15], which we included in a snappy-1.1.1.tar.gz file, and gnuplot-qt, which in included in gnuplot-4.4.3.tar file. In addition, the contents of the `uiHRDC` framework (downloaded from our repository) and copied into `/home/user/uiHRDC` directory of our docker image.
2. Connect to an instance of our docker image using `ssh/sftp`. Basically, we **expose** port 22 and create a user `'user'` with password `'userR1'` who can connect via `ssh`. This will allow the reader to connect to the docker container as to any remote server and to retrieve the final report by sftp. Once connected, `user` has `sudo` priviledges and, for example, can become `root` simply entering *sudo su*.

[15] `https://github.com/google/snappy`

3. Run `doAll.sh` script to automatically run all our experiments and generate our final report. This script must be run by `root` user. Note that we have installed (via apt-get) `screen` virtual terminal so that the user can disconnect from the docker container and still keep `doAll.sh` script running.

The minimum hardware requirements to run `doAll.sh` script would be to have a machine with at least 32GB RAM (and 16GB swap) and around 200GB of free disk space (in the file system were Docker keeps its files[16]). In our machine, i7-8700K@3.70GHz CPU (6 cores/12 siblings) with 64 GB of DDR4@2400MHz memory and a 7200rpm SATA disk, it took around 40 hours to run all the experiments from `doAll.sh` script.

4.1. Running the experiments: step-by-step

Below we show all the commands needed to use Docker[17] to reproduce our framework within a Docker container, then run the experiments within that container, and retrieve the final report (including also the gnuplot-type result data files).

1. Create a temporal working directory (i.e. `$TMP`) and move to it: `mkdir $TMP` and `cd $TMP`
2. Clone the `uiHRDC` GitHub repository at `https://github.com/migumar2/uiHRDC/`. It will create a directory `$uiHRDC`. Please rename it as `$DIR`. Then move to directory `$DIR`. Within `$DIR` you will find a file `Dockerfile` and two directories: `docker` and `uiHRDC`.

   ```
   $TMP> git clone https://github.com/migumar2/uiHRDC.git
   $TMP> mv uiHRDC $DIR
   $TMP> cd $DIR
   ```

3. Creating a docker image named `repet`: (you must be at `$DIR` directory)

   ```
   $DIR> sudo docker build -t repet --rm=true .        # Note: there is a final 'dot' (.)
   ```

4. Running a docker container named `repet-exp` that exports ssh/sftp port (22) via port 22222:

   ```
   $DIR> sudo docker run --name repet-exp -p 22222:22 -it repet
   ```

   Now, you can press `CTRL+P CTRL+Q` to detach from docker-container.
5. Connecting via ssh to the container and becoming `root`.

   ```
   $DIR> ssh user@localhost -p 22222     # enter password userR1 when prompted.
   $user@docker> sudo su          # enter password userR1 when prompted.
   $root@docker> screen -t EXPERIMENTS      # optional, to launch screen virtual terminal.
   ```

6. Move to the working directory `/home/user/uiHRDC` where `doAll.sh` script is located.

   ```
   $root@docker> cd /home/user/uiHRDC
   ```

[16] If you experiment space issues, a simple workaround could be to check which is the directory where Docker creates its images (e.g. `/home/shared/docker/tmp/` or `/var/lib/docker`), and to replace it by a symbolic link pointing to a location in a larger partition/disk.

[17] To install Docker, please refer to the installation guide for your host operating system: `https://docs.docker.com/install/`. In our Ubuntu system it simply consisted in running "`sudo apt-get install docker.io`"

7. Now we run `doAll.sh` script. `$root@docker> sh doAll.sh`
   and wait until `doAll.sh` had completed.
8. If you want to retrieve the final report and the results (gnuplot-type data files) via sftp you must grant access to those files to user `user`.
   `$root@docker>tar czvf /home/user/uiHRDC/report.tar.gz /home/user/uiHRDC/benchmark`
   `$root@docker>chown user:user /home/user -R`
9. Now we can close the ssh session (or detach from the docker container).
   `$root@docker> exit`
   `$user@docker> exit`
10. Now we can connect by sftp or scp using again port 22222 to download `/home/user/uiHRDC/report.tar.gz` from the docker container.
    `$DIR> sftp -P 22222 user@localhost:/home/user/uiHRDC` # enter password `userR1` when prompted.
    `$sftp> get report.tar.gz`
    or alternatively:
    `$DIR> scp -P 22222 user@localhost:/home/user/uiHRDC/report.tar.gz .` # use password `userR1`
    When decompressed, you will find the report here `$DIR/benchmark/report/report.pdf` and all the gnuplot data files within `$DIR/benchmark/report/figures` directory.
11. Finally you can stop `repet-exp` container, and if needed remove it and also `repet` image:[18]
    `$DIR> sudo docker stop repet-exp`
    `$DIR> sudo docker rm repet-exp`
    `$DIR> sudo docker rmi repet`

5. Conclusions and Future Work

We have briefly described all the techniques used in our original paper. In total, we had twenty two variants of posting list representations available that were used to create both non-positional and positional inverted indexes. In addition, we had six self-indexing techniques. Since those techniques have their own configuration parameters, and in some cases dependencies of libraries/software, it would be hard to replicate the results and to reuse our techniques by simply reading our original paper and cloning the (GitHub) repository were we had made our sources available. To overcome those limitations, this paper includes a detailed description of our replication framework `uiHRDC`. An interested reader could find not only our source code, our document collections and the query patterns used in our experiments, but also a set of related scripts. Those scripts permit us to replicate all our experiments with little effort and, additionally, generate

[18] The reader can use `sudo docker ps` and `sudo docker images` to see respectively the active containers and existing images.

a PDF report (using python, gnuplot, and latex) that contains the figures with the experimental results from our parent paper. In addition, we provide some configuration files to create a Docker container that reproduces our test environment (including all the required dependencies), and instructions regarding how to start the container, run the experiments, and finally download a copy of the final report from the Docker container. We hope that the descriptions and instructions provided along this paper will simplify the work of any reader interested in reusing the experiments from our original paper.

An interesting line of future work is to redesign `uiHRDC` to facilitate that new indexes to be added to the framework. The resulting benchmark framework would ensure reproducible experimentation for ongoing research about compression and indexing of highly repetitive collections.

6. Revision Comments

We would like to thank the authors for providing this valuable reproducibility platform, which allows both reproducing the results of its parent paper and encouraging the evaluation of new indexes for highly repetitive document collections in an easy and systematic way. Undoubtedly, this will be a topic of active research in the coming years given, for example, the cheapening of gene sequencing technology. Hence, the public availability of this type of benchmarking platform is becoming imperative. For this reason, we sincerely expect that uiHRDC sets an experimental standard for this line of research.

On the other hand, this work confirms again that the production of reproducible science is a difficult task, thus reproducibility of any research work must be considered and planned since the very beginning stages of development. Despite the reviewers reached a consensus about the reproducibility of the paper, the rigorous review process raised some reproducibility issues that left us some significant lessons on the difficulties of producing fully reproducible science and developing such ambitious reproducible platform as introduced herein. Next, we discuss our experience with this work as well as the lessons learned from our review.

We fully support the choice of using Docker for this kind of reproducibility experiments. Using a container makes it extremely simple to reproduce the experimentation framework, which otherwise would require to install the dependencies and compile the tested codes manually. However, review process unveiled other issues on the set-up and running of the uiHRDC experiments, which were fixed by providing specific instructions solving them. For instance, authors added information in the output report to warn the experimenter in those cases in which the experiments failed by lack of resources, as well as to ensure that the folder used by Docker resides in a drive with enough memory, which might not be the case for the default drive used by Docker.

In addition to the aforementioned set-up and running issues, some others arose with the experimentation itself in a first review: (1) a significant mismatch in the ordering of time-axis results derived from the

uncertainty in the evaluation of running time values; (2) a difference in the compression ratios obtained for the OPT-PFD indexing method; (3) missing values for Vbyte-Lzend and QMX-SIMD methods derived from software problems which did not provide any warning to the experimenter; and (4) differences of scaling between the original figures and those generated by uiHRDC, which made a detailed checking process difficult. Subsequently, all experimentation issues detailed above were either fixed or honestly and rigorously justified in the paper in a sample of best research practices. First, time measurement was significantly improved by increasing the number of evaluations with the aim of reducing its fluctuation. It worths to highlight that special attention should be put in any research for the reproducibility of running time values and their conclusions. Second, a few code bugs were fixed and the output experimentation report was extended to provide detailed information on the execution of the experiments. This further information shown to be very useful to guide the review process, and also to detect possible bugs for particular execution environments. Finally, axis scaling was revised to match the original paper exactly, and a bug with gnuplot was fixed so the generated report is produced with the same symbology of the parent paper. These later two issues highlight the importance of the presentation of the results to properly communicate research findings in a clear way.

The uiHRDC framework is a first try of standardizing experiments that, in absence of such a framework, would need to be repeated each time a new index is introduced in the literature. Thus, uiHRDC will be a very valuable resource to the research community by avoiding this tedious and costly task, which is also prone to errors.

As forthcoming activities, we invite the authors to extend uiHRDC with the aim of removing some drawbacks that hinder the integration of new indexes in their platform. The main drawback is that adding a new index is not trivial and requires editing several scripts; thus, any improvement in this sense will contribute to turn uiHRDC into a standard for comparison by future researchers.

Acknowledgments

This paper is funded in part by European Union's Horizon 2020 research and innovation programme under the Marie Sklodowska-Curie grant agreement No 690941 (project BIRDS). Antonio Fariña is funded by Xunta de Galicia/FEDER-UE [CSI: ED431G/01 and GRC: ED431C 2017/58] and by MINECO-AEI/FEDER-UE [ETOME-RDFD3: TIN2015-69951-R]. Miguel A. Martínez-Prieto is funded by MINECO-AEI/FEDER-UE [Datos 4.0: TIN2016-78011-C4-1-R]. Gonzalo Navarro is funded by the Millennium Institute for Foundational Research on Data (IMFD), and by Fondecyt Grant 1-170048, Conicyt, Chile.

References

[1] ACM. Artifact Review and Badging. `https://www.acm.org/publications/policies/artifact-review-badging`, 2018.

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

[3] R. A. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval*. Addison-Wesley Publishing Company, 2nd edition, 2011.

[4] F. Chirigati, R. Capone, R. Rampin, J. Freire, and D. E. Shasha. A Collaborative Approach to Computational Reproducibility. *Information Systems*, 59:95–97, 2016.

[5] F. Claude, A. Fariña, M. A. Martínez-Prieto, and G. Navarro. Universal Indexes for Highly Repetitive Document Collections. *Information Systems*, 61:1–23, 2016.

[6] F. Claude, A. Fariña, M. Martínez-Prieto, and G. Navarro. Compressed q-gram indexing for highly repetitive biological sequences. In *Proc. 10th International Conference on Bioinformatics and Bioengineering (BIBE)*, pages 86–91, 2010.

[7] F. Claude, A. Fariña, M. Martínez-Prieto, and G. Navarro. Indexes for highly repetitive document collections. In *Proc. 20th ACM International Conference on Information and Knowledge Management (CIKM)*, pages 463–468, 2011.

[8] C. Collberg, T. Proebsting, and A. M. Warren. Repeatability and Benefaction in Computer Systems Research: A Study and a Modest Proposal. Technical Report TR 14-04, University of Arizona, 2015.

[9] J. S. Culpepper and A. Moffat. Efficient set intersection for inverted indexing. *ACM Transactions on Information Systems*, 29(1):article 1, 2010.

[10] A. Fariña, M. A. Martínez-Prieto, F. Claude, and G. Navarro. *uiHRDC (Mendeley Data v1)*. 2018. `http://dx.doi.org/10.17632/xxntkjvtxw.1`.

[11] A. Fariña, N. Brisaboa, G. Navarro, F. Claude, A. Places, and E. Rodríguez. Word-based self-indexes for natural language text. *ACM Transactions on Information Systems*, 30(1):article 1, 2012.

[12] N. Ferro, N. Kando, N. Fuhr, M. Lippold, K. Jrvelin, and J. Zobel. Increasing Reproducibility in IR: Findings from the Dagstuhl Seminar on "Reproducibility of Data-Oriented Experiments in e-Science". *ACM SIGIR Forum*, 50:68–82, 2016.

[13] N. Ferro and G. Silvello. Rank-Biased Precision Reloaded: Reproducibility and Generalization. In *Proceedings of the 37th European Conference on IR Research, ECIR*, page 768780, 2015.

[14] J. He, J. Zeng, and T. Suel. Improved index compression techniques for versioned document collections. In *Proc. 19th ACM International Conference on Information and Knowledge Management (CIKM)*, pages 1239–1248, 2010.

[15] S. Heman. *Super-scalar database compression between RAM and CPU-cache*. PhD thesis, Centrum voor Wiskunde en Informatica (CWI), Amsterdam, 2005.

[16] S. Kreft and G. Navarro. On Compressing and Indexing Repetitive Sequences. *Theoretical Computer Science*, 483:115–133, 2013.

[17] N. Larsson and A. Moffat. Offline Dictionary-Based Compression. pages 296–305. IEEE Computer Society, 1999.

[18] J. J. Lastra-Díaz, A. García-Serrano, M. Batet, M. Fernández, and F. Chirigati. HESML: A scalable ontology-based semantic similarity measures library with a set of reproducible experiments and a replication dataset. *Information Systems*, 66:97 – 118, 2017.

[19] D. Lemire, L. Boytsov, and N. Kurz. Simd compression and the intersection of sorted integers. *Software: Practice and Experience (published online)*, 2015.

[20] J. Lin, M. Crane, A. Trotman, J. Callan, I. Chattopadhyaya, J. Foley, G. Ingersoll, C. Macdonald, and S. Vigna. Toward Reproducible Baselines: The Open-Source IR Reproducibility Challenge. In *Proceedings of the 38th European Conference on IR Research, ECIR*, pages 408–420, 2016.

[21] V. Mäkinen, G. Navarro, J. Sirén, and N. Välimäki. Storage and Retrieval of Highly Repetitive Sequence Collections. *Journal of Computational Biology*, 17(3):281–308, 2010.

[22] A. Moffat and L. Stuiver. Binary interpolative coding for effective index compression. *Information Retrieval*, 3(1):25–47, 2000.

[23] G. Navarro. *Compact Data Structures – A practical approach*. Cambridge University Press, 2016.

[24] G. Ottaviano and R. Venturini. Partitioned elias-fano indexes. In *Proc. 37th International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR)*, pages 273–282, 2014.

[25] K. Sadakane. New Text Indexing Functionalities of the Compressed Suffix Arrays. *Journal of Algorithms*, 48(2):294–313, 2003.

[26] G. K. Sandve, A. Nekrutenko, J. Taylor, and E. Hovig. Ten Simple Rules for Reproducible Computational Research. *PLOS Computational Biology*, 9:1–4, 2013.

[27] A. A. Stepanov, A. R. Gangolli, D. E. Rose, R. J. Ernst, and P. S. Oberoi. SIMD-based decoding of posting lists. In *Proc. 20th ACM International Conference on Information and Knowledge Management (CIKM)*, pages 317–326, 2011.

[28] F. Transier and P. Sanders. Engineering basic algorithms of an in-memory text search engine. *ACM Transactions on Information Systems*, 29:article 2, 2010.

[29] A. Trotman. Compression, simd, and postings lists. In *Proc. 19th Australasian Document Computing Symposium (ADCS)*, pages 50–57. ACM, 2014.

[30] P. Vandewalle, J. Kovacevic, and M. Vetterli. Reproducible Research in Signal Processing - What, Why, and How. *IEEE Signal Processing Magazine*, 26:37–47, 2009.

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

[32] I. Witten, A. Moffat, and T. Bell. *Managing Gigabytes*. Morgan Kaufmann, 2nd edition, 1999.

[33] A. Wolke, M. Bichler, F. Chirigati, and V. Steeves. Reproducible experiments on dynamic resource allocation in cloud data centers. *Information Systems*, 59:98–101, 2016.

[34] H. Yan, S. Ding, and T. Suel. Inverted index compression and query processing with optimized document ordering. In *Proc. 18th International Conference on World Wide Web (WWW)*, pages 401–410, 2009.

[35] M. Zukowski, S. Heman, N. Nes, , and P. Boncz. Super-scalar RAM-CPU cache compression. In *Proc. 22nd International Conference on Data Engineering (ICDE)*, page 59, 2006.

A. Compression Ratios

This appendix shows the exact compression ratios for all the techniques included in our `uiHRDC` framework. The corresponding tuning parameters are provided for each case ($\times$ is used for those techniques that do not require any parameter). Those values are included in Table 3.

Note that in the positional scenario, when the source text collection is compressed with Re-Pair, the parent paper allowed the extraction of snippets consisting of either 80 or $13,000$ chars. We considered sampling parameter values $sample\_ct = \{1, 2, 8, 32, 64, 256, 4096\}$. Table 4 shows the compression ratios of the RePair-compressed text that corresponds to each sampling configuration. Yet, the plot included in the parent paper (Fig.10-left) when 80 chars included only points for $sample\_ct = \{1, 2, 8, 32, 64\}$, and when extracting $13,000$ chars (see Figure 3, or Fig.10-right in the parent paper) we tuned $sample\_ct = \{1, 8, 32, 256, 4096\}$.

B. Using Self-Indexes for other types (non-document oriented) of Highly-Repetitive Collections

While self-indexes like `WCSA` or `WSLP` are designed to operate on words, the remaining approaches in our benchmark can effectively operate on an universal scenario; i.e. they are able to exploit repetitiveness

Non-positional indexes		
Method	**Compression ratio**	
	Value	**Parameterization**
Vbyte	4.4592%	×
VbyteB	3.3522%	$lenBitmapDiv = 8$
Vbyte-CM	4.4956%	$k = 32$
	4.7095%	$k = 4$
Vbyte-CMB	3.3754%	$k = 32, lenBitmapDiv = 8$
	3.4996%	$k = 4,\ lenBitmapDiv = 8$
Vbyte-ST	4.5147%	$B = 128$
	4.8650%	$B = 16$
Vbyte-STB	3.3820%	$B = 128, lenBitmapDiv = 8$
	3.5488%	$B = 16,\ lenBitmapDiv = 8$
Rice	3.0807%	×
RiceB	3.2235%	$lenBitmapDiv = 8$
Simple9	0.9130%	×
PforDelta	0.9372%	$pfdThreshold = 100$
QMX	4.8825%	×
Rice-Runs	0.3247%	×
Vbyte-LZMA	0.2030%	$minbcssize = 10$
Vbyte-Lzend	0.1420%	$ds = 256$
	0.1437%	$ds = 64$
	0.1508%	$ds = 16$
	0.1790%	$ds = 4$
Re-Pair	0.1040%	×
RePair-Skip	0.1097%	$repairBreak = 4\times10^{-7}$
RePair-Skip-CM	0.1195%	$k = 64,\quad repairBreak = 4\times10^{-7}$
	0.1212%	$k = 1,\quad repairBreak = 4\times10^{-7}$
RePair-Skip-ST	0.1279%	$B = 1024,\ repairBreak = 4\times10^{-7}$
EF-opt	0.4984%	×
OPT-PFD	0.7015%	×
Interpolative	0.5573%	×
varintG8IU	5.3144%	×

Positional indexes		
Method	**Compression ratio**	
	Value	**Parameterization**
Vbyte	35.1530%	×
Vbyte-CM	35.3805%	$k = 32$
	36.5899%	$k = 4$
Vbyte-ST	35.4935%	$B = 128$
	37.4281%	$B = 16$
Rice	36.7974%	×
Simple9	36.6233%	×
QMX	136.8762%	×
Vbyte-LZMA	9.7539%	$minbcssize = 10$
RePair	20.0641%	×
RePair-Skip	21.3769%	$repairBreak = 5 \times 10^{-7}$
RePair-Skip-CM	20.0786%	$k = 64,\quad repairBreak = 5 \times 10^{-7}$
	20.8584%	$k = 1,\quad repairBreak = 5 \times 10^{-7}$
RePair-Skip-ST	21.7456%	$B = 256,\ repairBreak = 5 \times 10^{-7}$
EF-opt	30.5630%	×
OPT-PFD	29.7424%	×
Interpolative	29.8820%	×
varintG8IU	37.9975%	×
RLCSA	2.4735%	$sample = 2048$
	2.8686%	$sample = 1024$
	3.6630%	$sample = 512$
	5.2489%	$sample = 256$
	8.4048%	$sample = 128$
	14.6924%	$sample = 64$
	27.2996%	$sample = 32$
WCSA	8.8921%	$\langle sA, sAinv, sPsi\rangle = \langle 2048, 2048, 2048\rangle$
	9.4071%	$\langle sA, sAinv, sPsi\rangle = \langle 512, 512, 512\rangle$
	11.0718%	$\langle sA, sAinv, sPsi\rangle = \langle 128, 256, 128\rangle$
	15.8673%	$\langle sA, sAinv, sPsi\rangle = \langle 64, 128, 32\rangle$
	17.9458%	$\langle sA, sAinv, sPsi\rangle = \langle 32, 64, 32\rangle$
	25.6777%	$\langle sA, sAinv, sPsi\rangle = \langle 16, 64, 16\rangle$
	50.8565%	$\langle sA, sAinv, sPsi\rangle = \langle 8, 8, 8\rangle$
SLP	3.2374%	×
WSLP	2.8962%	×
LZ77-index	1.8357%	×
LZend-index	2.3894%	×

Table 3: Summary of compression ratios for all indexes in the framework.

Compression ratio	Parameterization
1.306%	sample_ct = 1
1.265%	sample_ct = 2
1.227%	sample_ct = 8
1.215%	sample_ct = 32
1.213%	sample_ct = 64
1.211%	sample_ct = 256
1.210%	sample_ct = 4096

Table 4: Summary of compression ratios for the RePair-compressed version of the text with different sampling values.

underlying to any arbitrary sequence of chars.

Focusing on `SLP`, `LZ77-index`, and LZ-End, the implementations provided in `uiHRDC` can be easily used to self-index any data collection. Scripts for indexing and searching can be reused, although the number of parameters passed to `locate` changes. Currently, `locate` requires a parameter (called *doc_boundaries*) which sets the path to a file that contains the array of offsets that mark the beginning of each document in the text. It is required to map absolute positions to *(doc,offset)* pairs which indicate the position of each pattern occurrence within a document. However, this operation is not usually needed in a general scenario, so the corresponding parameter is neither needed.

Therefore, to invoke `locate` in a general scenario we have simply to remove *doc_boundaries*, and the script will deliver absolute positions of the pattern in the text.