Re-Pair Compression of Inverted Lists

Francisco Claude[*] Antonio Fariña[†] Gonzalo Navarro[‡]

Abstract

Compression of inverted lists with methods that support fast intersection operations is an active research topic. Most compression schemes rely on encoding differences between consecutive positions with techniques that favor small numbers. In this paper we explore a completely different alternative: We use Re-Pair compression of those differences. While Re-Pair by itself offers fast decompression at arbitrary positions in main and secondary memory, we introduce variants that in addition speed up the operations required for inverted list intersection. We compare the resulting data structures with several recent proposals under various list intersection algorithms, to conclude that the Re-Pair based variant offers an excellent space/time tradeoff, while retaining simplicity of coding.

1 Introduction

Inverted indexes are one of the oldest and simplest data structures ever invented, and at the same time one of the most successful ones in Information Retrieval (IR) for natural language text collections. It is a matter of looking at any book on the topic [4, 20], or to realize that they are at the heart of most modern Web search engines, where simplicity is a plus.

Essentially, an inverted index is a vector of lists. Each vector entry corresponds to a different *word* or *term*, and its list points to the *occurrences* of that word in the text collection. The collection is seen as a set of *documents*. The set of different words is called the *vocabulary*. Empirical laws well accepted in IR [12] establish that the vocabulary is much smaller than the collection size n, more precisely of size $O(n^\beta)$, for some constant $0 < \beta < 1$ depending on the text type.

Two main variants of inverted indexes exist [3, 22]. One is aimed at retrieving documents which are "relevant" to a query, under some criterion. Documents are regarded as vectors, where terms are the dimensions, and the values of the vectors correspond to the relevance of the terms in the documents. The lists point to the documents where each term appears, storing also the weight of the term in that document (i.e., the coordinate value). The query is seen as a set of words, so that retrieval consists in processing the lists of the query words, finding the documents which, considering the weights the query terms have in the document, are predicted to be relevant. Query processing usually involves somehow merging the involved lists, so that documents can get the combined weights over the different terms. Algorithms for this type of queries have been intensively studied, as well as different data organizations for this particular task [16, 20, 22, 1, 19]. List entries are usually sorted by descending relevance of the term in the documents.

The second variant, which is our focus, are the inverted indexes for so-called *full-text retrieval*. These are able of finding the documents where the query appears. In this case the lists point to the documents where each term appears, usually in increasing document order. Queries can be single words, in which case the retrieval consists simply of fetching the list of the word; or disjunctive

[*]David R. Cheriton School of Computer Science, University of Waterloo, Canada. `fclaude@cs.uwaterloo.ca`
[†]Dept. of Computer Science, University of A Coruña, Spain. `fari@udc.es`.
[‡]Dept. of Computer Science, University of Chile. `gnavarro@dcc.uchile.cl`.

queries, in which case one has to fetch the lists of all the query words and merge the sorted lists; or conjunctive queries, in which case one has to intersect the lists. While intersection can be done also by scanning all the lists in synchronization, it is usually the case that some lists are much shorter than the others [21], and this opens many opportunities of faster intersection algorithms. Those are even more relevant when many words have to be intersected.

Intersection queries have become extremely popular because of Google-like default policies to handle multiword queries. Given the huge size of the Web, Google gives priority to precision over recall, and queries are solved in principle by intersecting the inverted lists. Another important query where intersection is essential is the phrase query. This can be solved by intersecting the documents where the words appear and then postprocessing the candidates. In order to support phrase queries at the index level, the inverted index must store all the positions where each word appears in each document. Then phrase queries can be solved essentially by intersecting word positions. The same opportunities for smart intersection arise.

The importance of intersection algorithms for inverted lists is witnessed by the amount of recent research on the topic [10, 6, 2, 5, 7, 17, 9]. Needless to say, space is a problem in inverted indexes, especially if one has to store word positions. Much research has been carried out on compressing inverted lists [20, 15, 22, 9], and on its interaction with the query algorithms, including list intersections. Despite that algorithms in main memory have received much attention, in many cases one resorts to secondary memory, which brings new elements to the tradeoffs. Compression not only reduces space, but also transfers from secondary memory when fetching inverted lists (the vocabulary usually fits in main memory even for huge text collections). Yet, the random accesses used by the smart intersection algorithms become more expensive.

Most of the list compression algorithms rely on the fact that the inverted lists are increasing, and that the differences between consecutive entries are smaller on the longer lists. Thus, a scheme that represents those differences with encodings that favor small numbers work well [20]. Random access is supported by storing sampled absolute values, and some storage schemes for those samples considering secondary storage have been proposed as well [9].

In this paper we explore a completely different compression method for inverted lists: We use Re-Pair compression of the differences. Re-Pair [13] is a grammar-based compression method that generates a dictionary of common sequences, and then represents the data as a sequence of those dictionary entries. It is simple and fast at decompression, allowing for efficient random access to the data even on secondary memory, and achieves competitive compression ratios. It has been successfully used for compression of Web graphs [8], where the adjacency lists play the role of inverted lists. Using it for inverted lists is more challenging because several other operations must be supported apart from fetching a list, in particular the many algorithms for list intersection. We show that Re-Pair is well-suited for this task as well, and also design new variants especially tailored to the various intersection algorithms. We test our techniques against a number of the best existing compression methods and intersection algorithms, showing that our Re-Pair variants offer an attractive space/time tradeoff for this problem, while retaining simplicity.

2 Related Work

Intersection algorithms for inverted lists. The intersection of two inverted lists can be done in a merge-wise fashion (which is the best choice if both lists are of similar length), or using a set-versus-set (*svs*) approach where the longer list is searched for each of the elements of the shortest, to check if they should appear in the result. Either binary or exponential search are typically used for such task. The latter checks the list at positions $i + 2^j$ for increasing j, to find an element known to be after position i (but probably close).

Algorithm *bys* [5] is based on binary searching the longer list N for the median of the smallest list M. If the median is found, it is added to the result set. Then the algorithm proceeds recursively on the left and right parts of each list. At each new step the longest sublist is searched for the median of the shortest sublist. Results (comparing the number of required comparisons during intersection) showed that *bys* performs similarly to *svs* with binary search. As expected, both *svs* and *bys* improve *merge* algorithm when $|M| >> |N|$ (actually from $|M| \approx 20|N|$).

Multiple lists can be intersected using any pairwise *svs* approach (iteratively intersecting the two shortest lists, and then the result against the next shortest one, and so on). Other algorithms are based on choosing the first element of the smallest list as an *eliminator* that is searched for in the other lists (usually keeping track of the position where the search ended). If the eliminator is found, it becomes a part of the result. In any case, a new eliminator is chosen. Barbay et al. [7] compared four multi-set intersection algorithms: *i)* a pairwise *svs*-based algorithm; *ii)* an eliminator-based algorithm [6] (called *Sequential*) that chooses the eliminator cyclically among all the lists and exponentially searches for it; *iii)* a multi-set version of *bys*; and *iv)* a hybrid algorithm (called *small-adaptive*) based on *svs* and on the so-called *adaptive algorithm* [10], which at each step recomputes the list ordering according to their elements not yet processed, chooses the eliminator from the shortest list, and tries the others in order. Results [7] showed that the simplest pairwise *svs*-based approach (coupled with exponential search) performed best.

Data structures for inverted lists. The previous algorithms require that lists can be accessed at any given element (for example those using binary/exponential search) and/or that, given a value, its smallest successor from a list can be obtained. Those needs interact with the inverted list compression techniques.

The compression of an inverted list traditionally represents each original list $\langle p_1, p_2, p_3, \ldots, p_\ell \rangle$ as a sequence of d-gaps $\langle p_1, p_2 - p_1, p_3 - p_2, \ldots, p_\ell - p_{\ell-1} \rangle$, using a a variable-length encoding for these differences, for example γ-codes, δ-codes, Golomb codes, etc. [20]. More recent proposals [9] use byte-aligned codes, which lose little or no compression and are faster at decoding.

Intersection of compressed inverted lists is still possible using a merge-type algorithm. However, approaches that require direct access are not possible as sequential decoding of the d-gaps values is mandatory. This problem can be overcome by building an auxiliary index on the sequence of codes [9, 17]. The result is a two-level structure that consist of a top-level array that indexes the sequence of encoded values, and the indexed values themselves.

Assuming $1 \leq p_1 < p_2 < \ldots < p_\ell \leq u$, Culpepper and Moffat [9] extract a sample every $k' = k \log \ell$ values[1] from the compressed list, for a parameter k. Each of those samples and its corresponding offset in the compressed sequence is stored in the top-level array of pairs $\langle value, offset \rangle$ needing $\lceil \log u \rceil$ and $\lceil \log(\ell \log(u/\ell)) \rceil$ bits, respectively, while retaining random access to the top-level array. Searching for a value v into the compressed structure implies accessing the sample $\lceil v/k' \rceil$ and decoding at most k' codes. Results showed that intersection using *svs* coupled with exponential search in the samples performed just slightly worse than *svs* over uncompressed lists.

Sanders and Transier [17], instead of sampling at regular intervals of the list, propose sampling regularly at the domain values. The structure obtained is called a *lookup structure*. The idea is to

[1]Our logarithms are in base 2 unless otherwise stated.

create buckets of values identified by their most significative bits and building a top-level array of pointers to them. Given a parameter B (typically $B = 8$), and the value $k = \lceil \log(uB/\ell) \rceil$, bucket b_i stores the values $x_j = p_j \bmod 2^k$ such that $(i-1)2^k \leq p_j < i2^k$. Values x_j can also be compressed (typically using variable-length encoding of d-gaps). Comparing with the previous approach [9], this structure keeps only pointers in the top-level array, and avoids the need of binary-searching in it, as $\lceil p_j/2^k \rceil$ indicates the bucket were p_j appears. In exchange, the blocks are of varying length and more values might have to be scanned on average for a given number of samples. The authors also propose a refinement of the *svs* algorithm, where they keep track of up to where they have decompressed a given block in order to avoid repeated decompressions.

2.1 Re-Pair Compression Algorithm

Re-Pair [13] consists of repeatedly finding the most frequent pair of symbols in a sequence of integers and replacing it with a new symbol, until no more replacements are convenient. More precisely, Re-Pair over a sequence L works as follows: (1) It identifies the most frequent pair ab in L. (2) It adds the rule $s \rightarrow ab$ to a dictionary R, where s is a new symbol not appearing in L. (3) It replaces every occurrence of ab in L by s.[2] (4) It 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 (that is, proportional to its length), even if C is stored on secondary memory (as long as the dictionary of rules R is kept in RAM).

Re-Pair can be implemented in linear time [13]. However, this requires several data structures to track the pairs that must be replaced. This is problematic when applying it to large sequences. An approximate version [8] provides a tuning parameter that trades speed and memory usage for compression ratio. It achieves very good compression ratio within reasonable time and using little memory on top of the sequence (around 3%), and works well on secondary memory too.

Larsson and Moffat [13] proposed a method to compress the set of rules R. In this work we prefer another method [11], which is not so effective but 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 shape 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 and $o(|R_B|)$ bits of space [14]. 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. Nonterminals are shifted by the maximum terminal value to distinguish them. (See an example in Figure 1.)

3 Intersection of Re-Pair Compressed Inverted Lists

Data structure. Our basic idea is to differentially encode the inverted lists, transforming a sequence $\langle p_1, p_2, p_3, \ldots, p_\ell \rangle$ into $\langle p_1, p_2 - p_1, p_3 - p_2, \ldots, p_\ell - p_{\ell-1} \rangle$, and then apply the Re-Pair compression algorithm to the sequence formed by the concatenation of all the lists. A unique

[2]As far as possible, e.g., one cannot replace both occurrences of aa in aaa.

integer will be appended to the beginning of each list prior to the concatenation in order to ensure that no Re-Pair phrase will span more than one list. At the end of the compression process, we remove those artificial identifiers from the compressed sequence C of integers. We store a pointer from each vocabulary entry to the first integer of C that corresponds to its inverted list. We must also store the Re-Pair dictionary. The terminal symbols encode themselves their differential value.

This ensures that any list can be expanded in optimal time (i.e., proportional to its uncompressed size), as all the phrases that are expanded belong completely to the list. Moreover, if the dictionary is kept in main memory and the compressed lists on disk, then the retrieval requires accessing $\lceil \tilde{\ell}/B \rceil$ contiguous disk blocks, where B is the disk block size in integers and $\tilde{\ell} \leq \ell$ is the number of integers in the compressed list. Thus I/O time is also optimal in this sense.

Several intersection algorithms, as explained in Section 2, require other types of accesses to an inverted list: (a) at a given position in the list; (b) at a given value in the list (or the smallest successor of it that exists in the list). With Re-Pair compression there is no direct access to every list position, even if we knew its absolute value and a position in the compressed data. We can have direct access only to the Re-Pair phrase beginnings, that is, to integers in the compressed sequence. Accepting those "imprecisions" at locating, we can still implement the reviewed schemes for problems (a) and (b), yet in the case of sampling by domain [17] we would still need to represent the absolute sampled values, as our partition cannot be perfect.

On the other hand, and unlike any other compression method used, we can apply some skipping without sampling nor decompressing, by processing the compressed list symbol by symbol without expanding them. For this sake we need to know the phrase length of each dictionary symbol. This data can 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 length of each nonterminal. Terminals do not store this information but we identify them for having the smallest codes (and know they are of length 1).

This gives a rudimentary, yet pretty simple, method for doing accesses of type (a) to the inverted list: One scans the compressed list until exceeding the position sought. Then, the last symbol seen contains the position of interest. If it is a terminal, then we are done. Otherwise, one expands it into two new symbols. Using the size of the left symbol one knows whether to enter it or enter the right symbol. This continues until the first terminal to output is found, and then as many as desired following terminals are easily obtained. The time is proportional to the length of the compressed list plus the height of the tree, plus the number of list elements to output. The I/O cost is just the length of the compressed list divided by the block size.

The method is easily adapted to accesses of type (b). Instead of storing the number of terminals into which a nonterminal expands, we store the sum of the differences represented by those nonterminals. The terminal values themselves, on the other hand, are the difference they represent. Figure 1 illustrates a simple example. We note that, with this data associated to nonterminals, the final size is very close to that of an uncompressed representation, yet we have skipping information.

In both cases, we can obviously combine the skipping method with sampling and a two-level directory, yet our faster skipping allows for sparser sampling for the same time performance.

Another possibility, enabled by our fixed-length representation of the integers in C, is to use a bitmap B, aligned to the inverted list values, marking the phrase beginnings. Thus we could go directly to $C[rank_1(B, i)]$ in constant time instead of doing any skipping for accesses of type (a). The cost is low: one bit per list element, plus a term sublinear on that for *rank* queries [14]. In exchange, the sampling does not need to store positions in C, just absolute values.

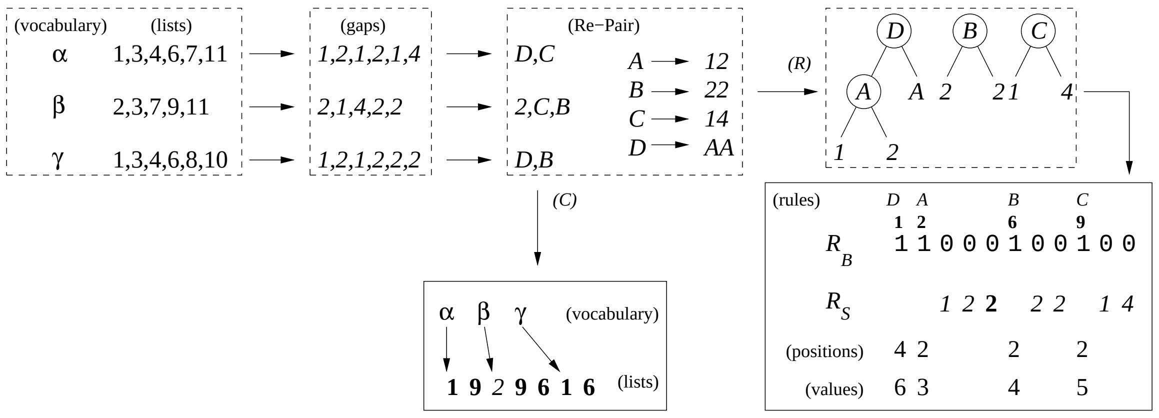

Figure 1: Example of inverted lists compressed with our Re-Pair-based method. Solid boxes enclose the data we represent. We include both variants of data aligned to the 1s in R_B. Bold numbers (nonterminals) in the lists and R_S refer to positions in R_B, whereas slanted ones (terminals) refer to gap values. To distinguish them, the maximum offset u is actually added to the bold numbers.

Intersection algorithm. As suggested above, we could implement any of the existing intersection algorithms on top of our Re-Pair compressed data structure. In this paper we will use two versions. A first does skipping without any sampling, but stores cumulative gap lengths for the nonterminals. A second version adds to the first a sampling of absolute values which is regular on the sequence of phrases (to avoid pointers, and because we expect to skip each phrase in constant time), and implements an *svs* strategy with sequential search in the samples [9], yet profiting from our skipping. To intersect several lists, we sort them in increasing order of their *uncompressed* length (which we must store separately). Thus we proceed iteratively, searching in step i the list $i+1$ for the elements of the *candidate* list. In step 1 this list is (the uncompressed form of) list 1, and in step $i > 1$ it is the outcome of the intersection at step $i-1$.

To carry out each intersection, the candidate list is sequentially traversed. Let x be its current element. We skip phrases of list $i+1$ (possibly aided by sampling), 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 $[x_1, x_2)$. We advance in the shorter list until finding the largest $x' < x_2$. We will process all the interval $[x, x']$ within the phrase representing $[x_1, x_2)$ by a recursive procedure: We expand nonterminals into their two components, representing subintervals $[x_1, z)$ and $[z, x_2)$. We partition $[x, x']$ according to z and proceed recursively within each subinterval, until we reach an answer (a terminal) to output or the interval $[x, x']$ becomes empty. Note, however, that our dictionary representation does not allow for this recursive partition. We must instead traverse their sequence in R_S and add up gaps. Nonterminals found in R_S, however, can be skipped.

4 Analysis

One can achieve worst-case time $O(m(1+\log \frac{n}{m}))$ to intersect two lists of length $m < n$ by different means, for example by binary searching the longer list for the median of the shortest and dividing the problem into two [2], or by exponentially searching the longer list for the consecutive elements of the shortest [9]. This is a lower bound in the comparison model, as one can encode any of the $\binom{n}{m}$

possible subsets of size m of a universe of size n via the results of the comparisons of an intersection algorithm, so these must be $\geq \log \binom{n}{m} \geq m \log \frac{n}{m}$ in the worst case, and the output can be of size m. Better results are possible for particular classes of instances of the problem [7].

We now analyze our skipping method, assuming that the derivation trees of our rules have logarithmic depth (which is a reasonable assumption). We expand the shortest list if it is compressed, at $O(m)$ cost, and use skipping to find its consecutive elements on the longer list, of length n but compressed to $n' \leq n$ symbols by Re-Pair. Thus, we pay $O(n')$ time for skipping over all the phrases. Now, consider that we have to expand phrase j, of length n_j, to find m_j symbols of the shortest list, $\sum_{j=1}^{n'} n_j = n$, $\sum_{j=1}^{n'} m_j = m$. Assume $m_j > 0$ (the others are absorbed in the $O(n')$ cost). In the worst case we will traverse all the nodes of the derivation tree up to level $\log m_j$, and then carry out m_j individual traversals from that level to the leaves, at depth $O(\log n_j)$. In the first part, we pay $O(2^i \log \frac{m_j}{2^i})$ for the 2^i binary searches within the corresponding subinterval $[x, x']$ of m_j at that level (an even partition of the m_j elements into $m_j/2^i$ is the worst case), for $0 \leq i \leq \log m_j$. This adds up to $O(m_j)$. For the second part, we have m_j individual searches for one element x, which costs $O(m_j(\log n_j - \log m_j))$. All adds up to $O(m_j(1 + \log \frac{n_j}{m_j}))$. Added over all j, this is $O(m(1 + \log \frac{n}{m}))$, as the worst case is $n_j = \frac{n}{n'}$, $m_j = \frac{m}{n'}$.

The overall complexity is thus $O(n' + m(1 + \log \frac{n}{m}))$, which exposes the need to use sampling to achieve the optimal worst-case complexity. One absolute sample out of $\log \frac{n}{n'}$ phrases in the lists would multiply the space by $1 + \frac{1}{\log \frac{n}{n'}}$ (which translates into a similar overall factor, in the worst case, when added over all the inverted lists), and would reduce the $O(n')$ term to $O(m \log \frac{n}{n'})$, which is absorbed by the optimal complexity as this matters only when $m \leq n'$. Recall also that the parse tree traversal requires that we do not represent the Re-Pair dictionary in compressed form.

5 Experimental Results

We focus on the incremental approach to solve intersections of sets of words, that is, proceeding by pairwise intersection from the shortest to the longest list, as in practice this is the most efficient approach [7, 9]. Thus we measure the intersection of two lists. From our technique (*repair*) we implemented the variants with and without sampling, as explained. Based on previous experiments [7, 9, 17], we compare with the following, most promising, basic techniques for list intersections (more sophisticated methods build orthogonally on these): *merge* (the merging based approach), *bin* and *seq* (the *svs* approach with binary and sequential search over the sampling of the longer list [9]), and *lookup* (*svs* where the sampling is regular on the domain and so the search on the samples is direct). We have used byte codes [9] to encode the differential gaps in all of the competing approaches, as this yields good space and excellent time performance.[3]

We have parsed collections FT91 to FT94 from TREC-4,[4] of 519,569,227 bytes (or 495.50MB), into its 210,138 documents (of about 2.4KB on average), and built the inverted lists of its 502,259 different words (a word is a maximum string formed by letters and digits, folded to lowercase), which add up to 50,285,802 entries. This small-document scenario is the worst for our *repair* index. We show also a case with larger documents formed by packing 10 of our documents into one; here the inverted lists add up to 29,887,213 entries. We first assume that the compressed lists fit in main memory and measure CPU times; later we consider lists resident on disk. Our machine is an Intel

[3]For different reasons (stability, publicness, uniformity, etc.) the competing codes are not available, so we had to reimplement all of them. For the final version we plan to leave all our implementations public.

[4]`http://trec.nist.gov`

Method	Vocabulary	Extra data	Inverted lists	Total
repair	2,699,656	(dict) 594,467	50,586,760	53,880,883
$(k = 1)$	3,955,308	+ 1,606,640	50,586,760	56,744,691
merge	1,569,568	—	59,406,108	60,975,676
bin,seq $(k = 2)$	4,456,556	8,606,784	55,668,305	68,732,645
$(k = 32)$	4,269,208	1,629,688	57,912,204	63,811,100
lookup $(B = 8)$	4,457,556	8,467,733	58,970,767	71,896,056
$(B = 64)$	4,269,208	1,170,400	59,355,998	64,795,606

Table 1: Space usage in bytes under the different schemes. Extra data refers to the dictionary space in case of *repair*, and to sampling for the others. Both can be controlled at will, but both are essential even to decompress (as sampled data is not replicated).

Core 2 Duo T8300, 2.4GHz, 3 MB cache, 4 GB RAM, running Ubuntu Linux, and compiled with gcc with full optimization. We used Re-Pair construction with parameter $k = 10,000$ [8], which takes just 1.5 min to compress the collection (slower compression could achieve even better ratios).

Table 1 shows the space requirements of the different schemes. Our *repair* with no sampling achieves 13% better compression than difference coding with no sampling, hence we can use a denser sampling for the same space (and even less)[5]. The dictionary is negligible and it would fit in RAM for very large collections, even if it scaled linearly with the data. The total space is about 10% of the text size and, discounting vocabulary, about 25% of the plain integer representation of inverted lists. When packing 10 documents into one, *repair* total space improves even in relative terms: it is about 5% of the text, discounting vocabulary it is 20% of an integer inverted list representation, and it is 27% better than difference coding.

We now consider intersection time. The outcome of the comparison significantly depends on the ratio of lengths between the two lists [17]. Thus we present results between randomly chosen pairs of words, as a function of this ratio. This time, however, the results for *repair* will depend on the absolute length of the lists, and thus we present results for two length ranges of the longer list. We generated 1,000 pairs per plot and repeated each search 1,000 times; then we grouped and averaged by coarse quotient between lengths.

Figure 2 shows the results. We chose variants using least space (but they are still much larger than ours). Our *repair* without sampling performs more or less as *merge* and with sampling it performs similarly to the best among *bin* and *seq* (*seq* implements the same intersection strategy on byte codes). The results improve for *repair* on the shorter lists. For the final version (for lack of time) we will implement strategies *bin* and *lookup* over *repair* phrases, as these perform better than *seq* for the same space. Thus we expect to achieve their performance while using much less space.

Re-Pair compression produces an interesting phenomenon on the lengths of the lists. The longer lists involve smaller and more repetitive differences, and thus they compress much better; Figure 3 illustrates the resulting length. Yet, expanding those resuling short lists is costly; this is why we consider the expanded length to decide the order of processing the lists.

Let us now consider the number of I/Os. We assume that the vocabulary, the samplings, and the Re-Pair dictionary fit in RAM. All are small and can be controlled at will (recall Table 1), so

[5]Yet this sampling is measured over phrases, so in terms of the original lists it twice as sparse as the competing ones with $k = 2$. Here our ability to skip over whole phrases in constant time is crucial.

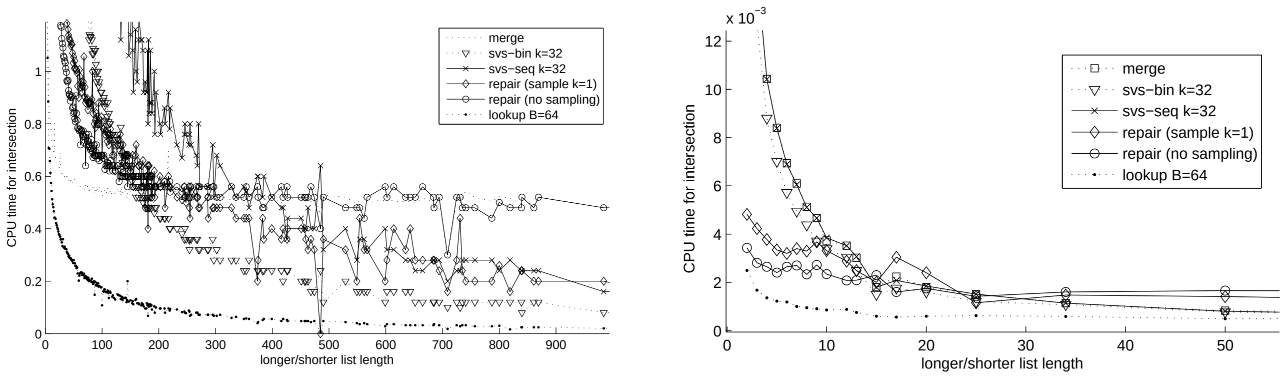

Figure 2: The intersection times as a function of the length ratios, for two different sizes of the longer list (left: 100,000 entries; right: 100 entries). Times are in microseconds.

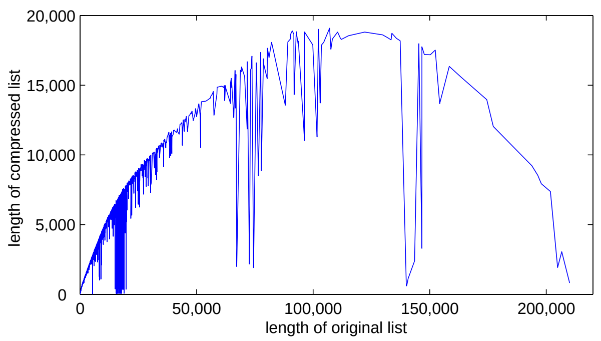

Figure 3: The compressed list sizes as a function of their uncompressed size, in entries.

the assumption is realistic. In this case the I/O cost of *repair* should be better because it achieves better compression ratio, and thus on average it has to load fewer disk pages to main memory. We will give some numbers in the final version.

6 Conclusions

We have presented a novel method to compress inverted lists. While previous methods rely on variable-length encoding of differences, we use Re-Pair on the differences. The compression we achieve is not only much better than difference encoding using byte codes (which permits denser sampling for the same space), but it also contains implicit data that allows for fast skipping on the unsampled areas. Thus our method achieves a good space/time tradeoff. In addition, we expect our index to perform well on secondary memory.

Byte coding is not the most space-efficient way to encode differences. In our experiments they achieved 56.65MB for the collection, whereas *repair* achieved 48.81MB. Gamma coding [20] gives 51.02MB, Delta coding gives 53.98, and Rice coding gives 35.47MB. The latter allows for much less space, and it translates into 4 times denser sampling for the same space. Yet, this does not compensate for the much slower decoding of bitwise codes compared to byte codes [18].

Future work involves comparing with more search techniques and data representations, including other text types, considering word-addressing indexes, other variants of our *repair* index (bitmap, sampling). We also aim at compressed Re-Pair dictionary representations that allow descending in the parse tree, and at researching on Re-Pair variants for compression of other types of inverted indexes, such as those for relevance ranking.

References

[1] V. Anh and A. Moffat. Pruned query evaluation using pre-computed impacts. In *Proc. 29th SIGIR*, pages 372–379, 2006.

[2] R. Baeza-Yates. A fast set intersection algorithm for sorted sequences. In *Proc. 15th CPM*, LNCS 3109, pages 400–408, 2004.

[3] R. Baeza-Yates, A. Moffat, and G. Navarro. *Searching Large Text Collections*, pages 195–244. Kluwer Academic, 2002.

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

[5] R. Baeza-Yates and A. Salinger. Experimental analysis of a fast intersection algorithm for sorted sequences. In *Proc. 12th SPIRE*, pages 13–24, 2005.

[6] J. Barbay and C. Kenyon. Adaptive intersection and t-threshold problems. In *Proc. 13th SODA*, pages 390–399, 2002.

[7] J. Barbay, A. López-Ortiz, and T. Lu. Faster adaptive set intersections for text searching. In *Proc. 5th WEA*, pages 146–157, 2006.

[8] F. Claude and G. Navarro. A fast and compact Web graph representation. In *Proc. 14th SPIRE*, LNCS 4726, pages 105–116, 2007.

[9] J. Culpepper and A. Moffat. Compact set representation for information retrieval. In *Proc. 14th SPIRE*, pages 137–148, 2007.

[10] E. Demaine and I. Munro. Adaptive set intersections, unions, and differences. In *Proc. 11th SODA*, pages 743–752, 2000.

[11] R. González and G. Navarro. Compressed text indexes with fast locate. In *Proc. 18th CPM*, LNCS 4580, pages 216–227, 2007.

[12] H. Heaps. *Information Retrieval - Computational and Theoretical Aspects.* Academic Press, 1978.

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

[14] I. Munro. Tables. In *Proc. 16th FSTTCS*, LNCS 1180, pages 37–42, 1996.

[15] G. Navarro, E. Moura, M. Neubert, N. Ziviani, and R. Baeza-Yates. Adding compression to block addressing inverted indexes. *Inf. Retr.*, 3(1):49–77, 2000.

[16] M. Persin, J. Zobel, and R. Sacks-Davis. Filtered document retrieval with frequency-sorted indexes. *J. Amer. Soc. Inf. Sci.*, 47(10):749–764, 1996.

[17] P. Sanders and F. Transier. Intersection in integer inverted indices. In *Proc. 9th ALENEX*, 2007.

[18] F. Scholer, H. Williams, J. Yiannis, and J. Zobel. Compression of inverted indexes for fast query evaluation. In *Proc. 25th SIGIR*, pages 222–229, 2002.

[19] T. Strohman and B. Croft. Efficient document retrieval in main memory. In *Proc. 30th SIGIR*, pages 175–182, 2007.

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

[21] G. Zipf. *Human Behaviour and the Principle of Least Effort.* Addison-Wesley, 1949.

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