Effective Construction of Relative Lempel-Ziv Dictionaries

Kewen Liao
kewen.liao@unimelb.edu.au

Matthias Petri
matthias.petri@unimelb.edu.au

Alistair Moffat
ammoffat@unimelb.edu.au

Anthony Wirth
awirth@unimelb.edu.au

Department of Computing and Information Systems
The University of Melbourne, Australia

ABSTRACT

Web crawls generate vast quantities of text, retained and archived by the search services that initiate them. To store such data and to allow storage costs to be minimized, while still providing some level of random access to the compressed data, efficient and effective compression techniques are critical. The Relative Lempel Ziv (RLZ) scheme provides fast decompression and retrieval of documents from within large compressed collections, and even with a relatively small RAM-resident dictionary, is competitive relative to adaptive compression schemes.

To date, the dictionaries required by RLZ compression have been formed from concatenations of substrings regularly sampled from the underlying document collection, then pruned in a manner that seeks to retain only the high-use sections. In this work, we develop new dictionary design heuristics, based on effective construction, rather than on pruning; we identify dictionary construction as a (string) covering problem. To avoid the complications of string covering algorithms on large collections, we focus on *k*-mers and their frequencies. First, with a reservoir sampler, we efficiently identify the most common *k*-mers. Then, since a collection typically comprises regions of local similarity, we select in each "epoch" a segment whose *k*-mers together achieve, locally, the highest coverage score. The dictionary is formed from the concatenation of these epoch-derived segments. Our selection process is inspired by the greedy approach to the Set Cover problem.

Compared with the best existing pruning method, CARE, our scheme has a similar construction time, but achieves better compression effectiveness. Over several multi-gigabyte document collections, there are relative gains of up to 27%.

1. INTRODUCTION

Large collections of data and meta-data are critical information assets, and often arise during web-based activities. For example, internet service providers are increasingly being required to retain extensive meta-data logs; web search companies derive their corporate success from the web pages that they crawl and retain; and even corporate entities are required to retain email and document histories in order to meet governance and compliance requirements. Such data is typically text-based, typically voluminous, and typically accessed in very small units. It is thus an ideal target for bespoke compression techniques.

Dictionary-based compression has a long history, going back at least to the work of Storer, Lempel, and Ziv in the late 1970s. The Relative Lempel Ziv scheme [5, 8], usually abbreviated to RLZ, is a relatively recent member of this family. It uses a semi-static dictionary and static or adaptive codes to provide fast atomic decompression and retrieval of documents in a compressed large collection. The dictionary is selected from the text that is to be compressed and is communicated to the decoder via a secondary channel and an alternative compression regime. Typical dictionary sizes are up to 1% of the original text. Then, once the dictionary is in place, fixed- or document-based blocks of the original text are (greedily) factored relative to the dictionary, and represented as two streams, one of "offset" values, and one containing the corresponding "copy-length" integers. Finally, for each block of data, the two streams are coded, using either static representations for integers (binary, or variable-byte codes), or using a per-block adaptive mechanism. The two streams can be treated differently if so desired.

The key advantage of RLZ is that pseudo-random access to the compressed text is possible, since each block can be independently decoded; yet compression effectiveness is relatively high, since the dictionary is drawn (in some way) from the whole of the text being stored. Recent experimentation exploring the trade-off between access efficiency and compression effectiveness showed that – even with a relatively small RAM-resident dictionary – RLZ factorization and ZLib-based encoding of the integer streams is competitive against state-of-the-art dynamic compression schemes [12].

From the start, Storer [14], as well as Storer and Szymanski [15], identified dictionary selection as a key optimization problem for dictionary-based compression schemes. In many scenarios, choosing an optimal dictionary is NP-hard. Typically to date, RLZ dictionaries have been formed from concatenations of long segments (of the order of 1 kiB) sampled at regular intervals from the collection, then pruned appropriately [5, 6, 17]. The drawback of this approach is that pruning commences with only a subset of the full text available to it, and if a string is not in the initial sampled dictionary, it is not in the final dictionary either. In this paper, we develop new dictionary-design heuristics, based on direct and targeted construction, rather than pruning; our mechanisms are not limited by the choices made by the initial (regular) sampling process.

In particular, we view dictionary construction as a string-based covering problem, seeking the combination of strings likely to yield the most economical factorization. To avoid the complications of string-covering algorithms on large collections, we focus on *k*-mers and their frequencies. With a reservoir sampler, we efficiently identify the most common *k*-mers. Since a collection typically com-

WWW 2016, April 11–15, 2016, Montréal, Québec, Canada.
ACM 978-1-4503-4143-1/16/04.
http://dx.doi.org/10.1145/2872427.2883042.

prises regions of local similarity, we break the text into “epochs”; then, within each epoch, select a dictionary segment whose *k*-mers have maximal coverage. Concatenated, and with suitable updates to the coverage score function, these segments then comprise the dictionary. That is, as the representative segment from each epoch for the dictionary, rather than taking a regular sample of the first

Algorithm 1 Regular sampling

```
Let $P$ be an empty dictionary pool, $s$ the segment length, $n$ the length of the text, and $m$ be the dictionary size required.
$M \leftarrow m/s$
for $i \in [0..M-1]$ do
    add to $P$, the substring $C^s[i(n/M)]$
return pool $P$
```

Regular Sampling. As originally proposed [5], the RLZ dictionary is formed by taking regular samples from C. That process is described in Algorithm 1, where $C^z[p]$ represents the string of length z, starting at the p^{th} symbol of C, that is, $C[p, p+1, \ldots, p+z-1]$. In total, the m-byte dictionary is the concatenation of the M segments, each of s bytes, in the pool. Typical values are $n = 64$ GiB, $m = 64$ MiB, and $s = 1$ kiB, so that there are $M = 2^{16} = 65{,}536$ segments concatenated to form the dictionary, drawn at intervals of 1 MiB. That is, in each of the M *epochs*, of length n/M, one segment is chosen, from the beginning of the epoch, and added to the dictionary. (Note that for convenience we assume that n is a multiple of M and of s, and that m is a multiple of s, but nothing in our implementation requires these exact relationships.)

Picking regularly spaced segments is fast, and produces surprisingly effective dictionaries [5]. However, during the dictionary creation process, previously selected segments are not considered when selecting the next dictionary segment. This can lead to inter-segment similarities that do not make compression more effective, and lead to wasted space in the dictionary. Inter-segment similarity can be measured by quantifying the repetitiveness of the dictionary.

To quantify repetitiveness, we examine the distribution of longest common prefix (LCP) values, computed by suffix-sorting the dictionary. A large LCP value, x, implies that two lexicographically adjacent suffixes of the text have a large common prefix. If those two suffixes do not overlap (in location), then the dictionary contains a duplicate string of length x. Figure 1 shows the distribution of LCP values arising in a 256 MiB dictionary formed from regularly sampled 1 kiB segments drawn from a 64 GiB section of a web crawl (see Section 4 for details of file "CC"). There are more than 10,000 LCP values larger than 1,024 in the dictionary, indicating that there are many long repeated strings in the dictionary. Indeed, that there are consecutive sampled blocks that by chance are repeated even across segment boundaries suggests, for example, that CC contains very long highly repetitive components. These would span more than one sampled segment and the exact entry point (for sampling) into the repeated string would be unimportant. The largest LCP value, 5,119, suggests that there may be five consecutive segments in the dictionary all containing the same symbol.

Dictionary Pruning. Pruning was at the core of previous attempts to produce space-efficient dictionaries. After generating a larger-than-required regular sampled dictionary, the least useful strings – for some definition of "useful" – were removed. The pruning problem is difficult to analyze: there are $\Theta(m^2)$ candidate strings to remove, and the effect of any particular removal is not obvious. A simple heuristic is to examine the factorization of C on D, and then remove from D the bytes that are the target of very few factors. A risk with this approach is some very long – if infrequently occurring – factors might be disrupted.

In a follow-up to their original RLZ paper, Hoobin et al. [6] introduce *segment-level pruning*. To prune the dictionary to the required size, they remove from the dictionary the constituent segments that are least frequently the target of a factor; this strategy is known as REM. A risk here is that a 1 kiB segment is likely to be too large to be the unit of removal: there could be a heavily used substring inside such a large segment.

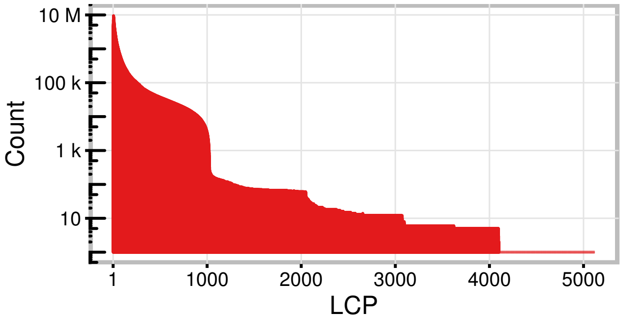

Figure 1: Distribution of LCP values of a 256 MiB dictionary created using regular sampling over a 64 GiB prefix of the file CC (described in Section 4).

The CARE Mechanism. Tong et al. [17] explore a more complex approach that estimates the cost of removing any given string from the dictionary. Given a set of strings that are candidates for removal from the dictionary, the ideal estimator would be to trial-compress the entire collection without each such string, then remove the string of least impact, then iterate greedily. But this level of recomputation is impractical. Instead Tong et al. rely on the dictionary to determine the contribution of a string, in a mechanism they refer to as being "contribution-aware reduction", or CARE. Each candidate string ς is factored against $D \setminus \varsigma$, the dictionary with ς removed. If there is a another copy of ς in D, then ς can be represented, with dictionary $D \setminus \varsigma$, as just one factor. If not, then ς might require several factors when parsed against $D \setminus \varsigma$. The measure for removal in CARE is the number of factors so generated, multiplied by the average target frequency of the bytes in ς, then divided by $|\varsigma|$.

The second part of CARE is the selection of candidate strings. They are determined greedily, left to right, and are selected to be of length at least some threshold λ, and with no byte having target frequency more than threshold ϕ. The thresholds λ and ϕ depend on both the collection and dictionary sizes: compressing a large collection using a small dictionary naturally increases the target frequency of bytes in the dictionary. Figure 2 shows a parameter exploration of two dictionaries (32 MiB and 128 MiB) of a 64 GiB prefix of a web crawl (file CC, described in detail in Section 4). For example, using the 32 MiB dictionary and thresholds $\lambda = 64$ and $\phi = 256$, only 4 MiB of candidate segments are identified; to prune the 32 MiB dictionary to 16 MiB, a frequency threshold of $\phi \approx 1024$ is required. However, at this point, ranking the candidate segments is inconsequential, as almost all segments are chosen in order to achieve the desired pruned dictionary size. Choosing thresholds that allow for meaningful candidate ranking for a given dictionary size, pruning size and text size is non-trivial.

As Tong et al. [17] suggest, an option might be to prune the dictionary via several rounds of the CARE heuristic, say reducing a 250 MiB dictionary first to 200 MiB, then to 150 MiB, 100 MiB, and finally 50 MiB. However, each CARE step requires that all of the frequency statistics be updated via re-factorization of C, meaning that multi-step CARE pruning has the potential to be very slow

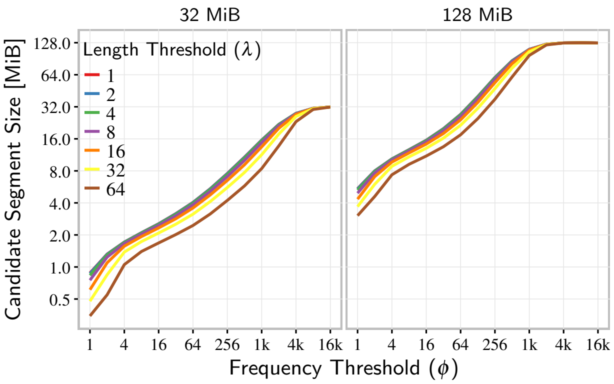

Figure 2: Pruning thresholds required by the CARE process to find candidate segments of a certain size for initial dictionaries of size 32 MiB (left) and 128 MiB (right), both extracted from the 64 GiB prefix of file CC.

for large datasets. Therefore, in our experiments, the principal baselines are REM and a single iteration of CARE.

In related work, Tong et al. [16] also explore what they term the "remote transmission" scenario, motivated by the search for efficient dictionaries for expanding, rather than fixed, collections. The question here is how best to construct a dictionary D_2 for a new tranche C_2 to be added to an existing collection C_1. Of course, both the remote and local hosts in a mirrored site already know the dictionary, D_1, that was previously constructed for C_1.

One technique suggested by Tong et al. is to factorize C_2 against the existing dictionary D_1, then build a subcollection C_2' from the concatenation of adjacent short factors, and sample a dictionary from C_2'. (In practice, a threshold for "short" that is twice the average factor length in this factorization works well.) The new auxiliary dictionary is then combined with D_1 to form a larger dictionary for the expanded collection. If the dictionary is not allowed to grow in this way, a second option is to produce D_2 directly from C_2, then prune the combined dictionary. In the transmission scenario, the locations that identify the pruned sections of D_1 are efficiently listed via a Golomb code (see Witten et al. [19]).

3. TARGETED CONSTRUCTION

We now introduce the design ideas for our dictionary-construction algorithms, including an analysis of the robustness of our scheme.

Sampling. Sampling is an essential component of big-data algorithms, so initially we explored alternatives to *regular* sampling. We first attempted to sample randomly from the whole collection; to simplify the selection, we focused on non-overlapping segments. Algorithm 2 generates a uniform random sample of M segments, without replacement, from the n/s adjacent, disjoint, length-s segments that comprise the collection. In each of the dictionary-construction schemes, we throw a family of segments into a "pool". The final step is to concatenate these pool segments after sorting them in "collection order". If the collection is processed sequentially, then the sorting step can be skipped.

Part of the rationale for regular sampling is that within a collection there are areas of local similarity. Therefore we also tried a stratified sample, where the stratification is along the length of the collection. Algorithm 3 chooses a single random segment from each epoch of length n/M.

Algorithm 2 Sample of non-overlapping segments

```
Let $P$ be an empty dictionary pool
$Segs \leftarrow [0..n/s-1]$
for $i \in [0..M-1]$ do
    uniformly at random, choose $J$ from $Segs$
    add to $P$, the substring $C^s[J \cdot s]$
    $Segs \leftarrow Segs \setminus \{J\}$
return pool $P$
```

Algorithm 3 Stratified sample of one segment per epoch

```
Let $P$ be an empty dictionary pool
for $i \in [0..M-1]$ do
    uniformly at random, from the range $[0..(n/M)-s-1]$,
        choose offset $j$
    add to $P$, the substring $C^s[i(n/M)+j]$
return pool $P$
```

Unfortunately, neither of these sampling schemes, alone, produced dictionaries that were much better than those of regular sampling. Seemingly, stratified or random sampling would only be better than regular sampling if there were some unfortunate "structure" or pattern in the collection; a regular sample would then fail to be a "reasonable" sample. Nevertheless, some of these sampling ideas inspired our covering techniques.

String Covering. The decision to include a substring in a dictionary must account for its length, its frequency, and its similarity to other substrings (already) in the dictionary. Forming a dictionary from segments of 1 kiB provides potential for longer matches. On the other hand, the discriminatory power of such segments is poor: only about 20 of every million strings of length 1 kiB appear more than once in the collection. Our compromise is to include in the dictionary these non-overlapping 1 kiB segments, but the criterion for inclusion is based on smaller substrings. These smaller substrings are in fact rolling k-mers. As shown in Section 4, letting $k = 16$, we can distinguish well between high and low-frequency k-mers. Overlapping k-mers avoid alignment issues, and allow for efficient rolling hash functions, such as Karp-Rabin [7]. With this segment/k-mer approach, we can apply covering algorithms inspired by the greedy approaches to Set Cover and Max Coverage [13].

Figure 3 shows the layout of our dictionary-construction scheme. It combines stratification and local maximal coverage. The collection is split into adjacent epochs, $E_0, \ldots, E_{M-1}$, with each epoch comprising n/m segments. In turn, each segment consists of $s-k+1$ overlapping k-mers. To estimate the frequency of each k-mer in the collection, a first pass through the collection builds a reservoir sample of k-mers, as detailed in the next subsection.

Accompanying Figure 3 is Algorithm 4, the dictionary-construction procedure. The algorithm iterates through the epochs either sequentially, as shown in Algorithm 4, or in random order, choosing one segment from each epoch. Every segment in the epoch is scored according to the frequencies of its k-mers, as described below. The highest-scoring segment is the one selected for the dictionary pool.

Segment Scoring. The *score* of a segment is a function of the *value* of all its $s-k+1$ rolling k-mers. The value of k-mer w is represented by $f(w)$: initially, it is (an estimate of) the frequency of w in the collection. However, since each k-mer is only needed

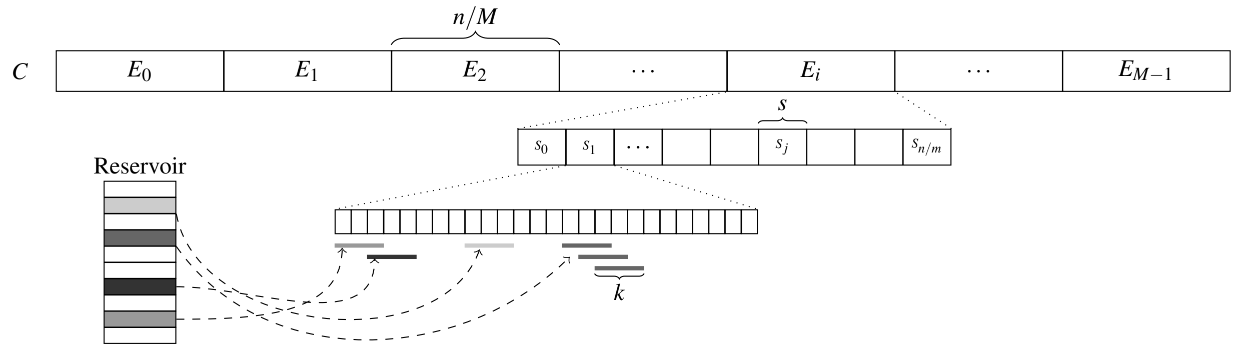

Figure 3: The local maximal covering procedure, where the collection C is split up into M epochs, $E_0 \dots E_{M-1}$. An epoch E_i consists of n/m disjoint segments of s bytes. To score each segment, rolling k-mers are compared to the reservoir of frequent k-mers. For each epoch, the highest scoring segment is added to the dictionary.

Algorithm 4 Local Maximal Coverage Dictionary (Sequential)

```
Let $P$ be an empty dictionary pool, and $g(\cdot)$ a segment-scoring function
for $i \in [0..M-1]$ do                          # for each epoch $E_i$
    for $j \in [0 \dots n/m-1]$ do                 # for each segment
        score the segment $S_j = C^s[i(n/M) + js]$
    add to $P$ the segment in $E_i$ with maximum $g(\cdot)$ score
    update the scoring function $g(\cdot)$
return pool $P$
```

once in the dictionary, $f(w)$ should be zero if w is already in the dictionary (pool). Moreover, we do not reward duplicate k-mers in a segment.

Therefore, let K_S be the *set* of k-mers $S^k[j]$, for $j \in [0..s-k]$, in segment S. To score a segment, we take the ℓ_p norm of the "vector" of k-mer frequencies: more precisely, $g(S) = (\sum_{w \in K_S} f(w)^p)^{1/p}$. The unweighted norm, ℓ_0, rewards long stretches of moderately frequent k-mers. The weighted norm ℓ_1 rewards highly frequent k-mers. And as shown in practice, $\ell_{1/2}$ is a useful compromise.

In Algorithm 4, the scoring function is updated after S is added to the dictionary pool: for each $w \in K_S$, $f(w)$ is set to zero. The string covering process is reminiscent of the greedy algorithm for Max Coverage, whose approximation ratio is 0.632 [13].

Reservoir Sampling. To cope with the full 426 GiB GOV2 collection, we involve several big-data tools. The scoring function relies on estimating k-mer frequencies. Our initial thought was to adopt the count-min sketch [1]. Unfortunately, we could not find a suitable compromise between sketch size and frequency accuracy. However, we only need good estimates for the frequencies of common k-mers. And since we can ignore rare k-mers in dictionary construction, a sample of the k-mers is appropriate; As shown below, the frequency estimates from the sample are sufficiently accurate for dictionary construction. (Indeed, a sample of the collection is an appropriate tool for other analyses and algorithms.)

A reservoir sample is simply a random sample taken without replacement from a population in an efficient way [9, 18]. Algorithms for constructing such samples often apply in streaming settings in which the population size is unknown. Typically, the algorithm starts with a prefilled sample of the right size, taken arbitrarily from the population. For every (other) element, the algorithm tosses biased coins to decide whether an incoming element should replace an element in the prefilled sample or be discarded. In our case, there are $n-k+1 \approx n$ rolling k-mers in C, which are the population elements. If we care most about those with frequency above t, then we create a sample of size $r = n/t$. Although the worst case running time of reservoir sampling is $O(n)$, the expected time can be designed to be $O(r(1+\log t))$, which is linear in the number of k-mers in the sample [18].

Our experiments confirm the sampling process is quite fast. Finally, to save a little extra space, a rolling Karp-Rabin hash is run over the k-mers. With an 8-byte hash, even in a 426 GiB collection, the chance of a k-mer hash collision is minimal, and scoring a segment requires $O(s)$ time.

Robustness. Consider a k-mer w that occurs $f(w)$ times in C. For each k-mer occurrence, the probability of inclusion in the reservoir sample is $1/t$. Therefore the expected number of occurrences in the sample is $f(w)/t$. So we let the (unbiased) estimate of the frequency of w, $\hat{f}(w)$, be the number of occurrences of w in the sample multiplied by t.

How good is this estimate? Multiple occurrences of the same k-mer in the sample are negatively correlated, so we can apply Chernoff bounds [11]. If the threshold t is 200, say, and if $f(w) = 100{,}000$, then the probability that $\hat{f}(w)/f(w) > 3/2$ is about $\exp(-500/12)$, which is about 10^{-18}. A similar estimate applies to $\Pr[\hat{f}(w)/f(w) < 1/2]$. Taking a union bound across all k-mers in a half-terabyte collection, the probability of at least one such mis-estimate is at most one in a million.

We mention in passing that we attempted a *global* maximum coverage dictionary construction scheme without imposing the epoch boundaries. Preliminary results were not promising, though further application of techniques for covering problems on massive datasets [2, 10, 13] may yet yield more competitive performance.

4. EXPERIMENTS

We evaluate the compression effectiveness of the new dictionary-creation strategies over three large-scale datasets, and compare them to previous RLZ algorithms. We first describe our experimental methodology including datasets, hardware, implementation and baselines.

Datasets. Three data sets are used in the experiments:

- The GOV2 standard information retrieval dataset of the TREC 2004 Terabyte Track, consisting of a test collection of $\approx$ 25 million documents crawled from the `.gov` subdomain in early 2004, which contains a mix of pdf and html

Name	Description
VLDB	The approach of Hoobin et al. [5], which uses regular sampling of 1 kiB segments to create the dictionary, and encodes offsets and lengths using ZLib (RLZ-ZZ in their paper).
AIRS	The improved encoding scheme of Petri et al. [12] (RLZ-ZZZ in their paper) which also uses regular sampling of 1 kiB segments for dictionary creation, but splits the encoding into three ZLib streams: offsets, lengths, and literals. Factors of length less than four are encoded as raw text.
REM	The AIRS scheme, coupled with the pruning mechanism of Hoobin et al. [4], which discards 1 kiB segments from the dictionary until the desired size is reached.
CARE	The AIRS scheme, coupled with the CARE pruning mechanism of Tong et al. [17], using initial thresholds $\phi = 10$ and $\lambda = 20$ and the FFT mode as suggested by Tong et al., but doubling ϕ until enough candidate segments are found to reach the desired dictionary size.
LMC	The AIRS scheme, coupled with our local maximum coverage-based dictionary creation method (Algorithm 4). Parameters that govern LMC include epoch-access order, either sequential (SEQ) or random (RAND); k-mer size; reservoir sampling threshold, t; and "power", p, in the ℓ_p-norm of segment-scoring function, $g(\cdot)$.
L+C	The LMC method coupled with the CARE pruning scheme, using the same CARE parameters discussed above.

Table 1: Baseline and methods used in our experimental evaluation.

documents, each truncated at 256 kiB. The entire dataset is 426 GiB uncompressed, and is denoted as Gov2-426.

- The CC data set, consisting of the first 100 files of the February 2015 Common Crawl (CC-MAIN-2015-11), corresponding to a 371 GiB subset of a recent 145 TiB webcrawl freely available at `commoncrawl.org`. The content was extracted using the tools from `https://bitbucket.org/hanzo/warc-tools`.
- The Kernel data set, consisting of the source code of all (332) linux kernel versions 2.2.X, 2.4.X.Y and 2.6.X.Y, downloaded from `kernel.org` and concatenated into one 78 GiB sequential file, appending the source packages in release-date order. This data set is highly repetitive, as only minor changes exist between subsequent kernel versions.

The Gov2 dataset serves as a benchmark in the compression literature [5, 6, 12, 17]. We adopt Gov2-426 as the large dataset in our experiments. For each dataset we additionally use a prefix of 64 GiB to explore the parameter space; these are referred to as Gov2-64, CC-64 and Kernel-64.

Hardware and Implementation. The parameter-setting experiments were run on a Intel Xeon E7-8837 CPU with 150 GiB RAM and a network file system which is part of a large computing cluster. The large-scale experiments were run on a Intel Xeon E5640 CPU using 148 GiB RAM and an SSD hard drive. Input and output files are memory mapped to allow the operating system (Ubuntu 15.04) to manage the available memory space. All methods are implemented in C++11 and compiled using Gcc 4.9.2 using all optimizations. The different index types were built using the SDSL library [3]. For fast factorization, we implemented a "flat" wavelet tree which uses one bitvector per symbol to efficiently perform searches in the compressed suffix array (CSA) over the dictionary. This increases the space usage of the factorization index, but the CSA is only built over the dictionary, which is small relative to the collection being compressed.

A fixed blocksize of 64 kiB was used throughout our experimentation, and blocks were compressed individually. Hoobin et al. [5] take individual documents to be blocks in their investigation, while Petri et al. [12] explore a range of blocksizes in their experiments. In terms of back-end coding, all of the results in this paper use the RLZ-ZZZ approach of Petri et al. – literals are generated to a third stream, whenever the factor length is less than four; and then all three streams are coded by passing arrays of integers to ZLib for conversion into bitstrings.

Our implementations and experimental framework are available at `https://github.com/unimelbIR/rlz-store`.

Baselines and Methodology. Table 1 lists the baselines and new methods considered as part of the experimental evaluation. Assuming the dictionary budget to be achieved is m, each of the pruning methods starts with an initial dictionary of size $2m$. All compression ratios reported are calculated as size of the uncompressed dictionary, plus the cost of all required meta data such as block addresses, plus the cost of compressed collection, all expressed as a percentage of the original text C. The *archive ratio* of Tong et al. [17] uses a standard compression tool such as `7zip` to compress the dictionary as well. The ratios reported below, which represent the *active* cost of the decompressor, are all slightly larger than the corresponding archive ratios. If a dictionary with $m/n = 0.5\%$ can be compressed to 30% of its original size, and if the collection can be compressed to 20% of its original size, then the archive ratio will be approximately 20.15%, versus an active ratio of 20.50%.

Parameter Exploration. We explored several properties of our dictionary-construction algorithm on 64 GiB prefixes of the datasets: (1) epoch access order; (2) k-mer size; (3) reservoir sampling threshold, t; (4) segment scoring function, $g(\cdot)$; and (5) segment size, s. These are discussed in the next several paragraphs.

Epoch Access Order. As our algorithm records the k-mers in the reservoir sample that have already been covered, processing the epochs sequentially can bias the selection of "useful" segments towards the beginning of C. To evaluate LMC, therefore, we fix $t = 256$, $k = 16$ and $s = 2$ kiB, and measure sequential processing (SEQ) and random-order epoch access (RAND).

Table 2 shows the compression obtained on the 64 GiB prefix Gov2-64 for $m \in \{16, 64, 256\}$ MiB and for two scoring functions, ℓ_0 and $\ell_{0.5}$ norms in $g(.)$. Random access order (RAND) consistently achieves better compression than sequential (SEQ). Interestingly, as m increases, the difference between the two methods also increases. This phenomenon is caused by the k-mer covering bias in SEQ. As epochs are processed, "valuable" (high frequency) k-mers are more likely to be covered by the epochs which are pro-

Method	Order	GOV2-64		
		16	64	256
LMC-ℓ_0	SEQ	19.89	17.27	15.22
	RAND	19.82	17.17	15.09
LMC-$\ell_{0.5}$	SEQ	19.48	17.14	15.19
	RAND	19.36	16.99	15.04

Table 2: Effect of SEQ and RAND epoch access modes on LMC, with segment-scoring norms of ℓ_0 and $\ell_{0.5}$, on the GOV2-64 GiB dataset for dictionaries of sizes 16 MiB, 64 MiB and 256 MiB. All values are active compression rates as percentages of the collection, including the uncompressed dictionary and all metadata.

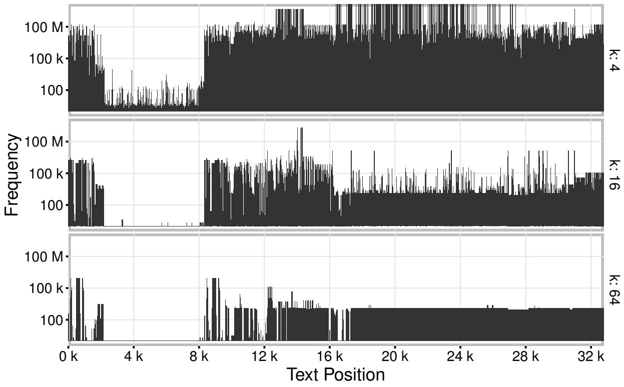

Figure 4: Frequency of k-mers for the first 32 kiB of the 64 GiB CC dataset for $k \in \{4, 16, 64\}$.

cessed earlier. However, epochs towards the end of the collection might contain globally the most valuable segments, but the algorithm might devalue them if they contain some already-covered k-mers. Assuming constant segment size, s, this effect is magnified as the dictionary size increases, as the epoch size is inversely proportional to the size of the dictionary. However, smaller epochs imply that more highly frequent k-mers will be covered by earlier epochs, increasing the coverage bias.

One possible drawback of the RAND approach is increased construction time. However, we found RAND to be only slightly slower than SEQ, taking less than 10% extra construction time, on both HDD and solid-state disk (SSD). Overall, since RAND achieves better compression effectiveness with only a small increase in construction time, all further results for LMC use RAND.

Varying k-mer Size. Figure 4 shows the frequency of each of the k-mers appearing in the first 32 kiB of the 64 GiB dataset CC-64 for $k \in \{4, 16, 64\}$. The maximum number of unique k-mers in a collection is σ^k, where σ is the size of the underlying alphabet. For small k, and large collections, a large fraction of all existing k-mers occur frequently. For example for $k = 4$, many of the k-mers that arise in the first 32 kiB of the dataset occur more than 100 million times across the full collection. When k is large, for example $k = 64$ as shown in the third graph of the figure, most k-mers occur infrequently. We found that when $k = 16$, across all our text collections, there is a reasonable spread of k-mer frequencies. In our experiments, each k-mer is hashed to a 8-byte word in $O(1)$ time using a Karp-Rabin hash function [7].

Varying Segment Size. For their algorithm VLDB, Hoobin et al. [5] chose a segment size of $s = 1$ kiB, noting that it worked well in practice. We explored a range of s values with VLDB, LMC and AIRS, and found that the performance of LMC is insensitive to s, whereas AIRS and VLDB become less effective as s increases. We conjecture that for AIRS, larger values of s incur more local string repetitions, whereas for LMC, those repetitions are accounted for in the design of the function $g(\cdot)$. We fix $s = 2$ kiB for LMC in the remaining experimentation, and fix $s = 1$ kiB for AIRS and VLDB.

Method	Threshold	GOV2-64		
		16	64	256
LMC-$\ell_{0.5}$	64	–	–	15.02
	128	19.36	16.99	15.02
	256	19.36	16.99	15.04
	512	19.37	17.01	15.06
	1024	19.40	17.04	–
	2048	19.45	17.07	–
	4096	19.49	–	–

Table 3: Effect of reservoir sample threshold, t, on active compression effectiveness, using the GOV2-64 dataset and dictionaries of sizes 16, 64, and 256 MiB.

Varying Reservoir Sampler Thresholds. The reservoir sampler filters out infrequent k-mers, which are less "valuable" to the dictionary. The reservoir sampler additionally reduces the size of the string covering problem. For example, if $t = 256$, then the expected number of k-mers in the reservoir sample is $\approx n/256$. In terms of space usage, for every 16-mer in the sample, eight bytes are needed to store its hash value, making the total sample size $n/32$ bytes.

To explore the impact of t on the effectiveness of LMC, we fix $s = 2$ kiB and $k = 16$. We additionally fix the scoring function to be based on the $\ell_{0.5}$ norm. Table 3 shows active compression effectiveness using the 64 GiB GOV2-64 dataset. For each dictionary size, the difference in compression ratios is relatively small, indicating that LMC is insensitive to the threshold value over a broad range. Our heuristic is to set $t = n/2m$ to allow sufficiently many useful k-mers to be stored in the reservoir. For a dictionary of size 64 MiB, t would be 512, and the corresponding results in Table 3 show that a smaller value of t barely improves compression effectiveness. However, for a dictionary of size 16 MiB, t is calculated to be 2048. In this case, for smaller t values, the compression effectiveness gradually increases, and we set an upper bound of 256 for t. That is, we set $t = \min\{n/2m, 256\}$. In terms of runtime, reservoir sampling algorithms can be implemented in time that is linear in the number of k-mer samples [9, 18]. In our experiments, for each of the 64 GiB datasets, sampling with $t = 256$ required less than 30 minutes; the sampling runtime decreases as t increases.

Varying Scoring Function. To test the impact of scoring functions based on different norms, we fix the dictionary size at $m =$ 64 MiB. The p values range from 0 to 2, the squared norm. Within a segment, the ℓ_0 norm equates to the total number of uncovered k-mers (at least identified by the reservoir sampler), while the ℓ_1 norm is the sum of the (estimated) frequencies of these k-mers. Figure 5 shows that on the GOV2-64 dataset, values of $p > 1$ achieve worse compression ratios as the norm value increases. The best values are between 0 and 1 and, overall, the $\ell_{0.5}$ computation performs best; this setting balances evaluating the total number of uncovered k-mers and with evaluating the sum of their frequencies. Table 4, which gives further active compression effectiveness scores, shows that for some combinations of datasets and dictionary size the ℓ_0

Method		Gov2-64			CC-64			Kernel-64		
		16	64	256	16	64	256	16	64	256
VLDB		21.58	18.89	16.57	–	–	–	–	–	–
AIRS		21.29	18.74	16.49	22.80	20.12	16.96	29.00	22.88	13.43
REM		21.19	18.42	16.05	22.67	19.87	16.33	28.77	22.12	11.00
CARE		20.35	17.88	15.68	21.87	19.13	15.47	28.09	21.61	9.85
LMC	ℓ_0	19.82	17.17	15.09	21.19	**17.66**	**14.06**	27.64	19.36	**7.22**
	$\ell_{0.5}$	**19.36**	**16.99**	15.04	**20.94**	17.69	14.11	**27.14**	**19.33**	7.27
	ℓ_1	19.58	17.07	15.06	21.18	17.94	14.19	27.48	19.64	7.36
L+C	$\ell_{0.5}$	19.47	**16.99**	**14.95**	**20.94**	17.67	14.19	27.17	19.63	8.47

Table 4: Active compression rates for three 64 GiB data sets using dictionaries of size 16 MiB, 64 MiB and 256 MiB, and the full range of RLZ variants discussed in this paper. Table 1 describes the settings used. The VLDB implementation was only executed on Gov2-64.

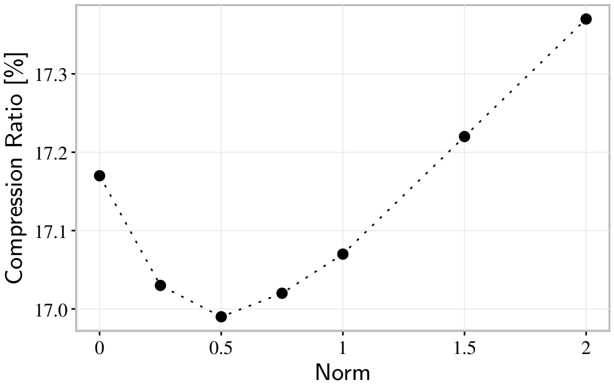

Figure 5: Effect of p in the ℓ_p norm of $g(\cdot)$ on compression effectiveness, for the 64 GiB Gov2 dataset with 64 MiB dictionaries.

Method	Gov2-426		
	64	256	1024
AIRS	12.92	11.45	10.17
CARE	12.14	10.74	9.47
LMC-$\ell_{0.5}$	**11.59**	**10.25**	**9.05**

Table 5: Active compression rates for the 426 GiB Gov2-FULL dataset for dictionaries of 64 MiB, 256 MiB, and 1024 MiB.

norm is slightly better than the $\ell_{0.5}$ norm. Since the difference is minor, we choose LMC-$\ell_{0.5}$ as the default scoring function for the remaining experiments.

Overall Comparison. Table 4 gives results for four previous RLZ variants for the three 64 GiB datasets, and the three dictionary sizes that we have been using throughout, seeking in part to establish that our experimental results are comparable to those reported in previous work [4, 5, 12, 17]. For Gov2-64, as reported by Petri et al. [12], AIRS achieves minor compression gains over VLDB. As it is outperformed by the more recent AIRS, we chose to not evaluate VLDB on CC-64 and Kernel-64; in addition, pruning using REM [4, 17] achieves a minor gain over AIRS for all dictionary sizes and datasets. Similarly CARE improves over REM by roughly 0.4% to 1.25%, which is similar to the gains reported by Tong et al. [17] for Gov2-100. On Gov2-100, Tong et al. [17] also report REM being outperformed by VLDB when the dictionary is halved, an effect we did not observe in our experiments.

The relative performance of all methods is similar for both of the webcrawl datasets (Gov2-64 and CC-64), which suggests that RLZ is consistent for this type of input. As a further reference point, ZLib (using -9) compressed the two webcrawl datasets to 20.84% and 19.48%, respectively, when applied in a per-block manner to the 64 kiB blocks used throughout our experimentation.

The Kernel-64 dataset compresses better than the webcrawl datasets with the large 256 MiB dictionary, but not with the smaller 16 MiB and 64 MiB dictionaries. This is because this dataset consists of different versions of linux kernel source code, appended together. There are only limited repetitions within each version of the source code, but it is highly repetitive across different versions. This dataset compresses to 22.78% using ZLib.

The performance of LMC is good for all scoring functions, with the $\ell_{0.5}$ norm performing best or close to best on all dataset and dictionary size combinations. The LMC approach also outperforms AIRS and all pruning methods, over all data sets and dictionary sizes, regardless of which scoring-function norm is used – that is, its superiority is not affected by the choice of norm. For the webcrawl datasets we observe an absolute improvements of LMC to AIRS of 1.5% to 2.8%, which correspond to relative gains of up to 17%. Similarly, the absolute improvements compared to CARE range from 0.5% to 1.44%, or relative advantages of up to 7.5%. For the Kernel-64 dataset, the relative compression improvement of LMC is much greater than for the webcrawl datasets. For Kernel-64 and a 256 MiB dictionary, LMC almost halves (with a 46% relative benefit) the space usage of the AIRS method, reducing the absolute compression ratio from 13.43% to 7.22%. In the same setting, LMC also outperforms CARE by 2.63%, a relative improvement of 27%. Combining CARE pruning with LMC (L+C) leads to minor improvements in one instance (Gov2-64 with 256 MiB dictionary) and is unhelpful in all other combinations. This suggests that CARE pruning fails to identify good candidate segments in LMC dictionaries.

Similar relativities are observed for the full 426 GiB Gov2 dataset, shown in Table 5. The LMC-$\ell_{0.5}$ approach achieves a 1.12% absolute improvement over AIRS for the 1 GiB dictionary (an 11% relative improvement), and is the best compression ratio reported for a compressor supporting efficient random access for this dataset. The same trend can be observed for the 64 MiB and 256 MiB dictionaries. As an overall statement of what we have achieved, observe that if AIRS is taken as a reference point, then the new LMC approach approximately doubles the improvements achieved by the previous CARE mechanism.

Figure 6: Average factor length per dictionary chunk of 256 kiB for the 64 GiB KERNEL dataset with three different 256 MiB dictionaries.

Dict.	Mean Factor Length			Literals (%)		
	AIRS	CARE	LMC	AIRS	CARE	LMC
collection = CC-64						
16	13.80	14.95	**15.42**	23.73	**20.56**	23.52
64	17.88	19.67	**20.63**	21.96	**20.54**	24.38
256	24.78	28.30	**30.49**	**22.51**	25.26	26.99
collection = KERNEL-64						
16	11.77	12.49	**12.77**	6.95	**4.84**	7.89
64	16.81	18.38	**20.00**	2.99	**1.81**	2.93
256	33.05	47.57	**64.79**	1.27	**0.80**	0.90

Table 6: Mean factor length and percentage of literals for AIRS, CARE and LMC-$\ell_{0.5}$ using $m \in \{16, 64, 256\}$ MiB and KERNEL-64 and CC-64.

Table 6 shows the mean (across blocks) average factor length and percentage of literals for AIRS, CARE and LMC-$\ell_{0.5}$, for KERNEL-64 and CC-64, and for three different dictionary sizes. Because the number of input bytes in each dataset is fixed, the mean factor length is an indicator of compression performance. Across all datasets, CARE produces longer factors than AIRS, while LMC in turn generates longer factors than CARE.

Discussion and Analysis. Figure 6 demonstrates that this improvement occurs uniformly across the whole dictionary. To construct the visualization, each of the 256 MiB dictionaries was split into $1{,}024 = 32 \times 32$ chunks, each of 256 kiB. The factors associated with each chunk were then tabulated as the original 64 GiB KERNEL-64 collection was being factored, and then, for each chunk, the average length of the associated factors was computed and plotted in a 32×32 grid (left to right and top to bottom). The deeper the shading, the greater the average factor length associated with that chunk. Note that while the relationship is not exactly one-to-one, corresponding cells across the three panes reflect groups of dictionary segments drawn from the same general parts of the original collection. Compared to the other two methods, the LMC approach generates longer factors right across the whole of the dictionary. There is also a clear pattern of repeated local variability in the lower third of each pane, which indicates the existence in that third of the collection of similar cycles of highly-repetitive content that is being identified and exploited by the RLZ paradigm.

The right-hand pane of Table 6 lists the percentage of factors that are represented as literals for the three methods; it provides a view that is in marked contrast to the observations made earlier about average factor lengths. While CARE produces shorter factors, it also tends to produce fewer literals than LMC. The webcrawl dataset exhibits a significantly larger number of literals than the more repetitive KERNEL dataset, and the efficient encoding of literals using a third stream has a non-trivial impact on overall compression effectiveness for webcrawl data.

Overall, LMC trades longer factors on average against degraded performance for short factors. Decreasing the number of literals produced, or identifying a better way of handling them, is a clear next challenge in terms of further improving the performance of RLZ systems on webcrawl data.

5. CONCLUSION AND FUTURE WORK

We have described a constructive approach to RLZ dictionary creation, and achieved substantially improved compression outcomes as a result. Rather than taking the first segment in each epoch and then pruning a too-large dictionary to get down to the required size, which is the approach used in previous work, we directly create a dictionary of the desired size by identifying the most useful segment to represent each epoch. The usefulness of a segment is estimated by aggregating scores from the k-mers comprising the segment. Across the set of large test files we investigated, consistent compression improvements are obtained, ranging from modest to very good, a clear endorsement of the new mechanism.

There are many avenues that might lead to further gains. For convenience and speed, epoch lengths have been taken to be uniform; however, the delineation of epochs could be allowed to depend on the collection content. In particular, relaxing the current presumption that the segment that represents each epoch is contiguous within the collection would allow segments to be constructed as a type of reassembly problem. In future work we plan to examine methods for constructing a synthetic segment that captures the nature of the text in a particular epoch, but without necessarily being a substring of that epoch.

Acknowledgment. This work was funded by the Australian Research Council's *Discovery Project* scheme (DP140103256), and by the Victorian Life Sciences Computation Initiative (VR0280), on its Peak Computing Facility at the University of Melbourne, an initiative of the Victorian State Government, Australia.

References

[1] G. Cormode and S. Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *J. Alg.*, 55(1): 58–75, 2005.

[2] G. Cormode, H. J. Karloff, and A. Wirth. Set cover algorithms for very large datasets. In *Proc. CIKM*, pages 479–488, 2010.

[3] S. Gog, T. Beller, A. Moffat, and M. Petri. From theory to practice: Plug and play with succinct data structures. In *Proc. SEA*, pages 326–337, 2014.

[4] C. Hoobin, S. J. Puglisi, and J. Zobel. Sample selection for dictionary-based corpus compression. In *Proc. SIGIR*, pages 1137–1138, 2011.

[5] C. Hoobin, S. J. Puglisi, and J. Zobel. Relative Lempel-Ziv factorization for efficient storage and retrieval of web collections. *PVLDB*, 5(3):265–273, 2011.

[6] C. Hoobin, S. J. Puglisi, and J. Zobel. Sample selection for dictionary-based corpus compression. In *Proc. SIGIR*, pages 1137–1138, 2011.

[7] R. M. Karp and M. O. Rabin. Efficient randomized pattern-matching algorithms. *IBM J. Res. Dev.*, 31(2):249–260, 1987.

[8] S. Kuruppu, S. J. Puglisi, and J. Zobel. Relative Lempel-Ziv compression of genomes for large-scale storage and retrieval. In *Proc. SPIRE*, pages 201–206, 2010.

[9] K.-H. Li. Reservoir-sampling algorithms of time complexity $O(n(1+\log(N/n)))$. *ACM Trans. Math. Soft.*, 20(4):481–493, 1994.

[10] C. L. Lim, A. Moffat, and A. Wirth. Lazy and eager approaches for the set cover problem. In *Proc. Aust. Comp. Sc. Conf.*, pages 19–27, 2014.

[11] A. Panconesi and A. Srinivasan. Randomized distributed edge coloring via an extension of the Chernoff-Hoeffding bounds. *SIAM J. Comp.*, 26(2):350–368, 1997.

[12] M. Petri, A. Moffat, P. C. Nagesh, and A. Wirth. Access time tradeoffs in archive compression. In *Proc. Asia Info. Retri. Soc. Conf.*, pages 15–28, 2015.

[13] B. Saha and L. Getoor. On maximum coverage in the streaming model & application to multi-topic blog-watch. In *Proc. SIAM Conf. Data Min.*, pages 697–708, 2009.

[14] J. A. Storer. NP-completeness results concerning data compression. Technical Report 234, Princeton University. Computer Sciences Laboratory, 1977.

[15] J. A. Storer and T. G. Szymanski. Data compression via textual substitution. *J. ACM*, 29(4):928–951, 1982.

[16] J. Tong, A. Wirth, and J. Zobel. Compact auxiliary dictionaries for incremental compression of large repositories. In *Proc. CIKM*, pages 1629–1638, 2014.

[17] J. Tong, A. Wirth, and J. Zobel. Principled dictionary pruning for low-memory corpus compression. In *Proc. SIGIR*, pages 283–292, 2014.

[18] J. S. Vitter. Random sampling with a reservoir. *ACM Trans. Math. Soft.*, 11(1):37–57, 1985.

[19] I. H. Witten, A. Moffat, and T. C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images, Second Edition*. Morgan Kaufmann, 1999.

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

[21] J. Ziv and A. Lempel. Compression of individual sequences via variable rate coding. *IEEE Trans. Inf. Th.*, IT-24(5):530–536, 1978.