Compact Auxiliary Dictionaries for Incremental Compression of Large Repositories

Jiancong Tong[†‡]
jctong@nbjl.nankai.edu.cn

Anthony Wirth[‡]
awirth@unimelb.edu.au

Justin Zobel[‡]
jzobel@unimelb.edu.au

[†]College of Computer and Control Engineering, Nankai University, China
[‡]Department of Computing and Information Systems, The University of Melbourne, Australia

ABSTRACT

Compression is widely exploited in retrieval systems, such as search engines and text databases, to lower both retrieval costs and system latency. In particular, compression of repositories can reduce storage requirements and fetch times, while improving caching. One of the most effective techniques is relative Lempel-Ziv, RLZ, in which a RAM-resident dictionary encodes the collection. With RLZ, a specified document can be decoded independently and extremely fast, while maintaining a high compression ratio. For terabyte-scale collections, this dictionary need only be a fraction of a per cent of the original data size. However, as originally described, RLZ uses a static dictionary, against which encoding of new data may be inefficient. An obvious alternative is to generate a new dictionary solely from the new data. However, this approach may not be scalable because the combined RAM-resident dictionary will grow in proportion to the collection.

In this paper, we describe effective techniques for extending the original dictionary to manage new data. With these techniques, a new auxiliary dictionary, relatively limited in size, is created by interrogating the original dictionary with the new data. Then, to compress this new data, we combine the auxiliary dictionary with some parts of the original dictionary (the latter in fact encoded as pointers into that original dictionary) to form a second dictionary. Our results show that excellent compression is available with only small auxiliary dictionaries, so that RLZ can feasibly transmit and store large, growing collections.

Categories and Subject Descriptors

E.4 [**Coding and Information Theory**]: [Data compaction and compression]; H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval—*Search process*

Keywords

Repository compression; dictionary representation; encoding; document retrieval

1. INTRODUCTION

Compression is a key component of large-scale information retrieval systems, such as web search engines [1, 4, 8, 21, 36]. It offers improved utilization of disk, efficient index processing, and, overall, an order-of-magnitude increase in volume of data that a single machine can handle. Compression of stored documents helps to improve caching in a single server [34] and to reduce the cost of replicating or transmitting collections between geographically separated data centers [11].

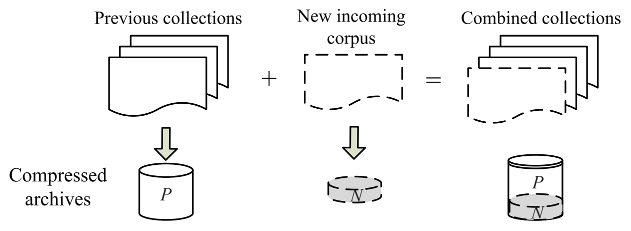
Compression of large text collections has been extensively studied for decades, but new approaches continue to appear [5, 9, 13, 15, 26]. Amongst these, relative Lempel-Ziv (RLZ) [15] is one of the most effective compression techniques for large repositories. RLZ offers both good compression effectiveness and extremely fast atomic retrieval of individual documents. It has the particular advantage that it assumes only that the material being stored contains redundancy in the form of repeated strings: there is no requirement, for example, that the collection consist of text in a certain language. As we describe later, RLZ exploits a RAM-resident dictionary that consists of a single long string, against which efficient encoding and decoding of a collection can be performed. In contrast to ad hoc methods based on `zlib`[1] and similar libraries, only the document of interest need be retrieved and decoded; in addition, decoding is dramatically faster.

Existing methods for repository compression often assume that the collection is static [9, 26]. However, in many situations, the data collection is dynamic; in this paper, we focus on the scenario of a growing data collection. We assume that the collection grows in large increments, or *tranches*, each of which is around a gigabyte, or perhaps even a terabyte. Although the new data may arrive in smaller increments, the retrieval system formally adds it to the collection (in compressed form), or transmits it, only when the threshold for a the tranche is reached. We refer to this tranche-by-tranche expanding collection as an *incremental collection*, and we refer to the problem of compressing an incremental collection as *incremental compression.*

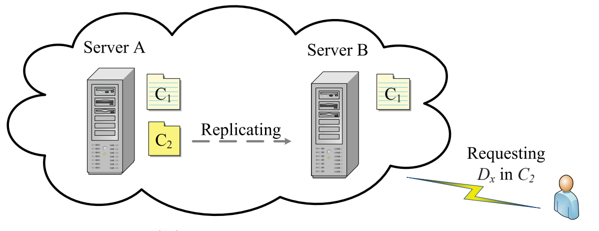
In this paper, we consider the following two common scenarios, each well served by incremental compression.

Local archiving: A single server stores an incremental collection; the goal is to compress each tranche as well as possible. We refer to this as the *updating* scenario, as depicted in Figure 1(a).

[1]`zlib.net`



(a) Updating scenario



(b) Transmitting scenario

Figure 1: Two incremental corpus compression scenarios.

Remote transmission: A primary server replicates its collection at a remote server, and thus must update the remote server after each tranche is added. The goal is to minimize the volume of *transmitted* data between the servers. We refer to this as the *transmitting* scenario, as shown in Figure 1(b).

In each of these scenarios, we desire fast per-document retrieval as well as a good compression ratio, and thus RLZ is a suitable choice. With the adoption of RLZ, however, the goals for these scenarios are slightly modified. When *updating*, we additionally wish to avoid excess memory consumption, and thus should limit the size of the dictionary. When *transmitting*, we additionally wish to avoid excess data transmission, and thus should represent the dictionary (for the new tranche), as well as the tranche itself, as compactly as possible.

In this paper, we propose that, to compress a new tranche, the dictionary for existing tranches be exploited selectively. As we discuss in the next section, considered alone, this existing dictionary may not compress a subsequent tranche well. However, a wholly new dictionary (or sequence of wholly new dictionaries) may require too much space. We therefore propose methods for creating a small auxiliary dictionary for each new tranche, based on the dictionary built for previous tranches. Our results show that excellent compression can be maintained with relatively limited additional space requirements. We also describe a highly efficient method—indeed, an optimal method, under reasonable assumptions—for compactly describing which parts of the existing dictionary are used in compressing the new tranche.

In summary, we show in this paper (i) that incremental compression in which previous tranches assist in compressing new tranches is not only possible, but also feasible; (ii) a principled method for compactly describing a subset of the fragments of a large file; and (iii) through our experimental results, that our new method informs better dictionary construction principles for the original RLZ algorithm. Together, these outcomes significantly extend the value and application of RLZ for practical compression of large collections.

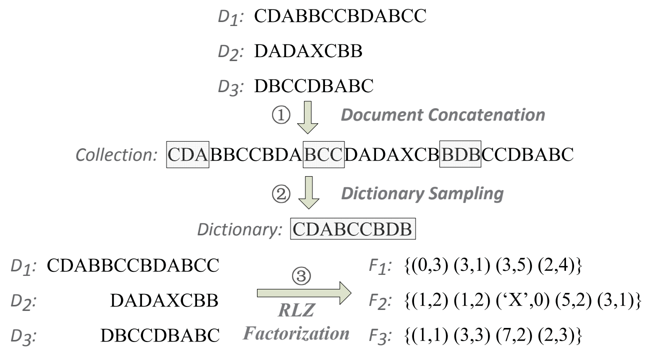


Figure 2: The RLZ scheme: concatenation, sampling, and factorization.

2. BACKGROUND AND RELATED WORK

In this section we review RLZ, previous work on pruning of RLZ dictionaries, and related work on compression of (incremental) collections.

2.1 Relative Lempel-Ziv

RLZ [15] is an effective dictionary-based compression algorithm for large collections (such as of web documents or genomes) where there are discoverable long repeated strings. In brief, the mechanism of RLZ is that a text dictionary (a continuous long string) is generated from the collection, which is then used to parse (*factorize*) the collection text into a sequence of substrings, each of which can be found in the dictionary. String matching is the key component of such relative Lempel-Ziv factorization [19], and can be accelerated by ancillary data structures, such as the suffix array [20].

Figure 2 illustrates the principles of RLZ. Consider a text collection comprising three documents, D_1, D_2, and D_3. The documents in the collection are concatenated into a long string [step ①], which is sampled at regular intervals to form an external, but RAM-resident, dictionary [step ②]. Each document is then factorized against the dictionary into a sequence of fragments [step ③], each of which is represented as a (position, length) *factor*, a pointer into the dictionary. The factors of each document are then encoded using standard byte-oriented codes. In this paper, following the settings in previous work [31], `ZLIB` encodes positions and `VBYTE` [32] encodes lengths.

RLZ provides excellent compression because random sampling is an effective way of collecting repeated strings. If a string is common, it is likely to be represented in the dictionary. (It is also likely to be present more than once in the dictionary, an issue that we examine under the topic of pruning, below.) The dictionary thus provides a highly effective resource against which the collection can be compressed.

RLZ provides fast random access to individual documents for two reasons. First, the dictionary is static (in contrast to the adaptive dictionaries used in the classic Lempel-Ziv (LZ) family [35]), and thus exists as a simple array of bytes. In current computer architectures, decoding a factor—which may contain hundreds of bytes—requires only a single copy instruction, and is thus extremely efficient. Second, during encoding, document boundaries are respected, so that the end of a document is always the end of a factor, and thus each individual document is a separate stored entity.

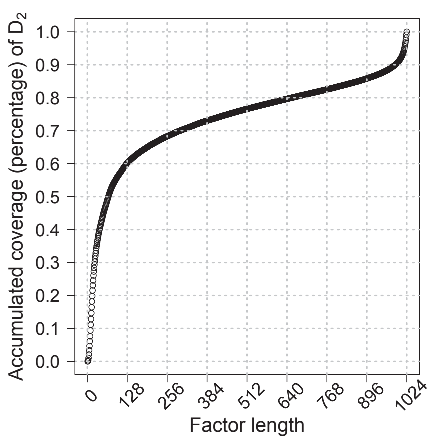


Figure 3: Redundancy between existing dictionary $\mathcal{D}_1$ and new dictionary $\mathcal{D}_2$. For clarity, factors whose length exceeds 1024 are omitted (they contribute only ∼0.77 MB of the 1 GB).

In this paper, we use the notation $\mathcal{F}(\mathcal{C}, \mathcal{D})$ to denote the sequence of factors produced by factorizing a collection $\mathcal{C}$ against a dictionary $\mathcal{D}$. The compressed size of $\mathcal{F}(\mathcal{C}, \mathcal{D})$ is represented as $|\mathcal{F}(\mathcal{C}, \mathcal{D})|_c$, while $\mathcal{R}(\mathcal{C}, \mathcal{D})$ denotes the corresponding compression ratio of RLZ: the ratio of $|\mathcal{F}(\mathcal{C}, \mathcal{D})|_c$ to the (uncompressed) collection size $|\mathcal{C}|$.

RLZ for growing collections. RLZ has been claimed to perform well on incremental collections [15], where the initial dictionary is generated from the first tranche of data. However, compared to a dictionary generated from the whole collection, we found in preliminary experiments that this compression effectiveness can degrade substantially.

We should therefore generate a dictionary from the data in the new tranche. An obvious solution is to sample a dictionary solely from this tranche. However, our preliminary investigations revealed that this new dictionary and the existing dictionary have much in common; ignoring the existing dictionary completely is therefore wasteful.

To illustrate this, the initial 50 GB of the GOV2 dataset [7] serves as the first tranche, with the next 50 GB of GOV2 serving as the second tranche. (In this paper, a gigabyte (GB) is 2^{30} bytes, with KB and MB similarly defined.) From each tranche, we generate a 1-GB RLZ dictionary: these are called $\mathcal{D}_1$ and $\mathcal{D}_2$. Finally, to identify the overlap between $\mathcal{D}_1$ and $\mathcal{D}_2$, we determine $\mathcal{F}(\mathcal{D}_2, \mathcal{D}_1)$. Figure 3 shows the cumulative coverage of $\mathcal{D}_2$ by common factors, as a function of factor length. Large factors indicate 'redundancy': these sections of $\mathcal{D}_2$ are covered well by $\mathcal{D}_1$. Figure 3 shows, for example, that 40 % of $\mathcal{D}_2$ is covered by strings in $\mathcal{D}_1$ (factors) whose length is at least 128 bytes.

We conclude that $\mathcal{D}_1$ is helpful for compressing the new tranche and that we need only a small auxiliary dictionary specific to the new tranche, provided it has little overlap with $\mathcal{D}_1$. These observations lead to the new methods in this paper.

Dictionary pruning. The original RLZ study [15] reported that, in many situations, a significant portion of dictionary is never referred to. We infer that the original scheme produces dictionaries with significant redundancy. Subsequent studies [16, 31] investigated pruning of the RLZ dictionary to reduce redundancy in the sampled dictionary, and therefore provide better compression. Hoobin et al. [16] proposed removing the blocks—fixed-size chunks, each say 1 KB—that are least frequently referred to. Tong et al. [31] subsequently introduced a pruning algorithm (called CARE) that selects candidate prunable segments from the dictionary and removes those segments that their heuristic suggests contribute least to compression effectiveness. Due to its demonstrated superiority, we implement the CARE algorithm for pruning in our experiments.

2.2 Collection compression methods

Compression techniques have been adopted in multiple basic components of information retrieval systems [9, 33]. Applications include compressed textual web documents [9], compressed full-text indexes [24], compressed inverted indexes [37], and compressed permuterm indexes [10]. Our concern here is with compression of the stored documents, for which RLZ is one of the most effective techniques.

String substitution. Common-string substitution is a typical method used to compress text collections [28]. To encode large repositories, long repeated strings are identified and then delta encoding is applied [26]. As a practical industrial example, Google adopted such a technique [2] for handling of long repeated strings in the collection data of their Bigtable system [6]. RLZ can also be regarded as an application of string substitution, though the substitutions refer to an external dictionary rather than to previous parts of the collection itself.

Versioned collections. Versioned collections—snapshots of Wikipedia at different times, for example, comprising multiple versions of each document—can be compressed effectively due to inter-version redundancy [13, 14]. In particular, delta compression, the core idea in which is to encode one file in terms of another, performs well [17, 18, 25]. Incremental corpus compression, as described in this paper, is a different problem. A new tranche does not necessarily comprise modified versions of material already in the collection. Although delta compression can also be applied to non-versioned collections, the inter-file dependencies inevitably slow document decompression and retrieval significantly.

Dynamic text collections. Incremental collection compression has been studied in a range of other contexts. Moffat et al. [23] applied a word-based compression scheme to semi-static compression. They proposed methods to help the model to handle new words that occur in the updates to the collection. With a model derived from a relatively small sample of the text corpus, their scheme can effectively compress a large text database. Brisaboa et al. [3] also described adaptive vocabulary schemes for incremental collections. However, in common with other schemes based on parsing of text into words and non-words, compression performance is relatively poor, decoding is not fast, and application is limited to specific kinds of data.

Hoobin et al. [15] argued that RLZ also works well in dynamic environments. The reported performance is reasonable, but, as we show in this paper, substantial improvements are available if alterations to the dictionary are allowed.

Remote replication. In the context of transmission of large repositories to remote servers, which is similar to our transmitting scenario, existing schemes usually incorporate de-duplication. They then use delta encoding to reduce the volume of transferred data [27, 29, 30]. As for versioned collections, de-duplication sets up inter-file dependencies, which makes atomic document retrieval much slower than it is in RLZ.

3. INCREMENTAL COMPRESSION FOR LOCAL ARCHIVING

In this section, we introduce a new algorithm for the updating scenario (Figure 1(a)). In the context of RLZ compression, the optimization problem is:

> Given a collection $\mathcal{C}_1$ with dictionary $\mathcal{D}_1$, a new tranche $\mathcal{C}_2$, and a memory budget B, generate an auxiliary dictionary $\mathcal{D}_\Delta$ that minimizes
>
> $$|\mathcal{F}(\mathcal{C}_2, \mathcal{D}_1 + \mathcal{D}_\Delta)|_c\,,$$
>
> while guaranteeing that $|\mathcal{D}_1| + |\mathcal{D}_\Delta| \le B$.

In the rest of the section, tranche $\mathcal{C}_2$ is compressed with a dictionary comprising auxiliary dictionary $\mathcal{D}_\Delta$ concatenated with $\mathcal{D}_1$. Combined dictionaries of this form will be denoted by $\mathcal{D}_\cup$: Table A.1 is a glossary for this section.

Because the size of the ancillary data structure for string search, for example a suffix array, is proportional to the dictionary size, we ignore this constant-factor overhead and focus solely on dictionary size when consider budget B.

3.1 Consulting the existing dictionary

The easiest way to generate an auxiliary dictionary, $\mathcal{D}_\Delta$, for the new tranche, $\mathcal{C}_2$, would be to sample only from $\mathcal{C}_2$. The *sampling* technique is the same as in the standard RLZ scheme (Figure 2). We regard this new tranche-only dictionary, $\mathcal{D}_2^s$, as a baseline for our study. Figure 3 suggests that this $\mathcal{D}_2^s$ would have much in common with $\mathcal{D}_1$. To derive an auxiliary dictionary $\mathcal{D}_\Delta$ from $\mathcal{D}_2^s$ with 'fresh' material, we therefore consult $\mathcal{D}_1$.

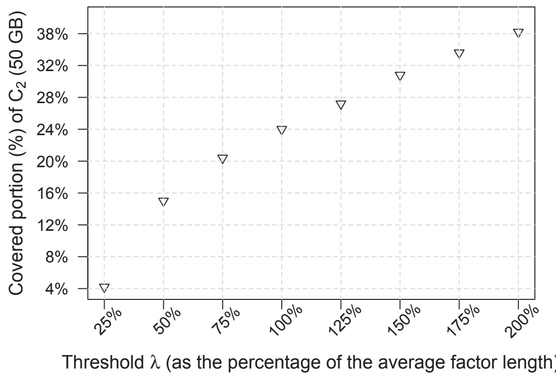
Dictionary upon dictionary (DuD). Our first attempt is to sample a large dictionary $\mathcal{D}'$ from tranche $\mathcal{C}_2$ (say of similar size to $\mathcal{D}_1$) and remove the long strings that it shares with $\mathcal{D}_1$. Obviously, we cannot remove all the redundancy between the dictionaries, because even a singleton byte in $\mathcal{D}'$ is a (redundant) copy of the same symbol in $\mathcal{D}_1$. Thus we only consider removing common substrings longer than some threshold λ; we find that the average factor length in $\mathcal{F}(\mathcal{D}', \mathcal{D}_1)$ is a good initial choice for λ. The remaining material in $\mathcal{D}'$ could serve as $\mathcal{D}_\Delta$, and thus its size may be controlled by the choice of λ. In preliminary experiments, we found that such $\mathcal{D}_\Delta$ leads to a minor improvement in compression over the commensurate baseline dictionary ($\mathcal{D}_2^s$). Since our subsequent innovations perform considerably better than this 'DuD' approach, we save space and omit these preliminary results.

Algorithm 1 CuD approach for generating $\mathcal{D}_\Delta$.

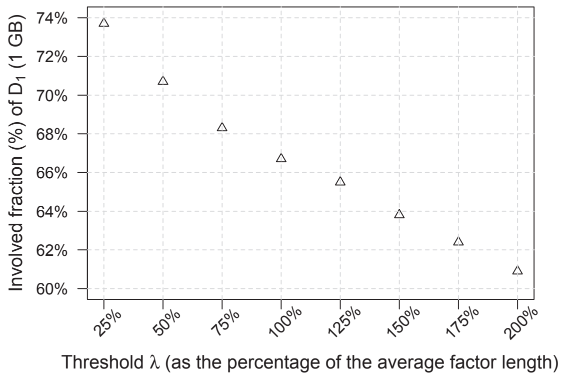
```
Input: Tranche $\mathcal{C}_2$, Existing dictionary $\mathcal{D}_1$, Threshold $\lambda$ for
    factor length, Memory budget $B$
Output: Auxiliary dictionary $\mathcal{D}_\Delta$ for $\mathcal{C}_2$ of size $\le B - |\mathcal{D}_1|$
1: $\mathcal{F} \leftarrow$ Factorize($\mathcal{C}_2, \mathcal{D}_1$)
2: $\mathcal{F}_\lambda \leftarrow \{f \mid f \in \mathcal{F}$ and $|f| \le \lambda\}$
3: $\mathcal{F}_\Delta \leftarrow$ factors in $\mathcal{F}_\lambda$ that are adjacent in $\mathcal{C}_2$ to other $\mathcal{F}_\lambda$
    factors.
4: $\mathcal{C}_2^L \leftarrow$ concatenation of factors in $\mathcal{F}_\Delta$
5: return $\mathcal{D}_\Delta \leftarrow$ Sample($\mathcal{C}_2^L, B - |\mathcal{D}_1|$)
```

Collection upon dictionary (CuD). The DuD algorithm performs poorly because its $\mathcal{D}'$ is sampled uniformly from $\mathcal{C}_2$. It therefore fails to capture sufficiently those segments of $\mathcal{C}_2$ that $\mathcal{D}_1$ struggles to encode, and leads to many small factors. Matching our intuition, experiments confirm that the presence of a large number of small factors in a factorization indicates poor compression. Our goal is thus to create a $\mathcal{D}_\Delta$ that leads to fewer (longer) factors. We therefore identify more carefully the parts of $\mathcal{C}_2$ that $\mathcal{D}_1$ encodes poorly: that is, those areas with many small factors. From $\mathcal{F}(\mathcal{C}_2, \mathcal{D}_1)$, we concatenate (runs of) the short factors (whose length is below threshold λ) to form $\mathcal{C}_2^L$. Algorithm 1 (CuD)'s final step is to sample $\mathcal{C}_2^L$ to derive the auxiliary dictionary $\mathcal{D}_\Delta$. The CuD functions `Factorize()` in line 1 and `Sample()` in line 5 refer to the synonymous RLZ procedures (Figure 2).

Short factors in $\mathcal{F}(\mathcal{C}_2, \mathcal{D}_1)$ indicate segments of $\mathcal{C}_2$ that are poorly compressed by $\mathcal{D}_1$. However, were we to add an *isolated* short factor to our new dictionary (one between two long factors), this would not really assist in compression as it would remain a short factor. Instead, as detailed in line 3 of Algorithm 1, we include only runs of *at least two* adjacent



(a) The portion of $\mathcal{C}_2$ that is covered by factors of $\mathcal{F}(\mathcal{C}_2, \mathcal{D}_1)$ of length **at most** λ.



(b) The fraction of $\mathcal{D}_1$ that comprises targets of the factors of $\mathcal{F}(\mathcal{C}_2, \mathcal{D}_1)$ of length **at least** λ.

Figure 4: The impact of length threshold λ. The initial and second 50 GB of Wikipedia dataset serve as $\mathcal{C}_1$ and $\mathcal{C}_2$, $|\mathcal{D}_1| = 1$ GB.

Table 1: Results of applying the CuD method for the updating scenario on 10-GB tranches from the initial 50 GB of GOV2. The threshold λ in each round is set to *twice* the average factor length of the corresponding $\mathcal{F}(\mathcal{C}_2, \mathcal{D}_1)$.

Updating round	B (MB)	$\lvert\mathcal{C}_1\rvert$ (GB)	$\lvert\mathcal{D}_1\rvert$ (MB)	$\lvert\mathcal{C}_2\rvert$ (GB)	$\lvert\mathcal{C}_2^L\rvert$ (GB)	$\lvert\mathcal{D}_2^\Delta\rvert$ (MB)	$\lvert\mathcal{D}_\cup^\Delta\rvert$ (MB)	$\mathcal{R}(\mathcal{C}_2, \mathcal{D}_\cup^\Delta)$ (%)	$\lvert\mathcal{C}_1 + \mathcal{C}_2\rvert$ (GB)	$\mathcal{R}(\mathcal{C}_1, \mathcal{D}_1)$ (%)	$\mathcal{R}_{total}$* (%)
0	1000	0	0	10	10.00	1000	1000	12.38	10	-	12.38
1	1250	10	1000	10	5.27	250	1250	13.67	20	12.38	13.03
2	1500	20	1250	10	5.31	250	1500	13.41	30	13.03	13.15
3	1750	30	1500	10	5.21	250	1750	13.08	40	13.15	13.14
4	2000	40	1750	10	5.17	250	2000	12.18	50	12.14	12.94

* $\mathcal{R}_{total} = (\lvert\mathcal{C}_1\rvert \times \mathcal{R}(\mathcal{C}_1, \mathcal{D}_1) + \lvert\mathcal{C}_2\rvert \times \mathcal{R}(\mathcal{C}_2, \mathcal{D}_\cup^\Delta))/(\lvert\mathcal{C}_1\rvert + \lvert\mathcal{C}_2\rvert)$

Table 2: Results of applying baseline method for the updating scenario on 10-GB tranches from the initial 50 GB of GOV2. (Note that $\mathcal{C}_2^L$ here is exactly the same as $\mathcal{C}_2$, because $\mathcal{D}_2^s$ is sampled from the entire $\mathcal{C}_2$.)

Updating round	B (MB)	$\lvert\mathcal{C}_1\rvert$ (GB)	$\lvert\mathcal{D}_1\rvert$ (MB)	$\lvert\mathcal{C}_2\rvert$ (GB)	$\lvert\mathcal{C}_2^L\rvert$ (GB)	$\lvert\mathcal{D}_2^s\rvert$ (MB)	$\lvert\mathcal{D}_\cup^s\rvert$ (MB)	$\mathcal{R}(\mathcal{C}_2, \mathcal{D}_\cup^s)$ (%)	$\lvert\mathcal{C}_1 + \mathcal{C}_2\rvert$ (GB)	$\mathcal{R}(\mathcal{C}_1, \mathcal{D}_1)$ (%)	$\mathcal{R}_{total}$* (%)
0	1000	0	0	10	10.00	1000	1000	12.38	10	-	12.38
1	1250	10	1000	10	10.00	250	1250	14.12	20	12.38	13.30
2	1500	20	1250	10	10.00	250	1500	13.83	30	13.30	13.46
3	1750	30	1500	10	10.00	250	1750	13.46	40	13.46	13.46
4	2000	40	1750	10	10.00	250	2000	12.51	50	13.46	13.28

* $\mathcal{R}_{total} = (\lvert\mathcal{C}_1\rvert \times \mathcal{R}(\mathcal{C}_1, \mathcal{D}_1) + \lvert\mathcal{C}_2\rvert \times \mathcal{R}(\mathcal{C}_2, \mathcal{D}_\cup^s))/(\lvert\mathcal{C}_1\rvert + \lvert\mathcal{C}_2\rvert)$

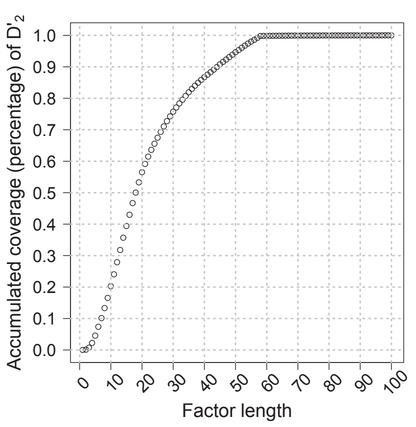


Figure 5: Redundancy between existing dictionary $\mathcal{D}_1$ and new dictionary $\mathcal{D}'_2$ sampled from $\mathcal{C}_2^L$. For clarity, factors whose length exceeds 100 are omitted (they contribute only ~26.8 KB of the 1 GB).

short factors in the new dictionary. Concatenated, these constitute helpful longer strings in the factorization of $\mathcal{C}_2$ against the new combined dictionary.

Figure 4 illustrates some properties of the relationship between $\mathcal{C}_2$ and $\mathcal{D}_1$ as a function of λ on the Wikipedia dataset (a 251-GB English Wikipedia snapshot extracted from the ClueWeb09 dataset[2]). As shown in Figure 4(a), even when allowing factors up to twice the average size to be candidates, less than 37 % of the content in $\mathcal{C}_2$ is considered as a sampling source for $\mathcal{D}_\Delta$. Referring to Figure 4(b), with a threshold λ that is twice the average factor length, only 61 % of $\mathcal{D}_1$ is involved in compressing the 63 % of $\mathcal{C}_2$ that is compressed 'well'. The well-compressed parts of $\mathcal{C}_2$ are those with long factors, those that are *not* picked to form $\mathcal{F}_\lambda$, and hence $\mathcal{C}_2^L$.

To illustrate the effectiveness of our approach, we consider the factor lengths of a CuD-like dictionary, factorized against the original dictionary $\mathcal{D}_1$. The cumulative distribution in Figure 5 is derived from the same tranches as in Figure 3. Here, however, we sample a 1-GB dictionary $\mathcal{D}'_2$ from the 22.1-GB $\mathcal{C}_2^L$ that emerged from the CuD algorithm. (In this analysis, we need not restrict the dictionary to the smaller size expected of an auxiliary dictionary.)

Figure 5 shows that the redundancy between $\mathcal{D}'_2$ and $\mathcal{D}_1$ is dramatically less than that between $\mathcal{D}_2$ and $\mathcal{D}_1$ in Figure 3. Indeed, there are no long factors, and so we have successfully captured more patterns from the poorly compressed portions of $\mathcal{C}_2$, on which $\mathcal{D}_1$ and $\mathcal{D}_2$ fail.

Table 3: Results of compressing against the 1000-MB *static* initial dictionary on 10-GB tranches from the initial 50 GB of GOV2, for the updating scenario.

Round	0	1	2	3	4
$\mathcal{R}(\mathcal{C}_2, \mathcal{D}^1)$ (%)	12.38	16.57	17.97	18.57	18.97
$\mathcal{R}_{total}$ (%)	12.38	14.48	15.64	16.37	16.89

3.2 Experimental results

We first show, on a relatively small dataset, that in practice our CuD method performs better than the $\mathcal{C}_2$-only baseline. We divide the initial 50 GB of the GOV2 collection into five 10 GB tranches; the first tranche serves as the initial collection (Round 0), and there are update Rounds 1, 2, 3, and 4.

With CuD, dictionary $\mathcal{D}_\cup^\Delta$ comprises a small $\mathcal{D}_2^\Delta$, generated by CuD algorithm, together with $\mathcal{D}_1$. For comparison, our baseline is $\mathcal{D}_\cup^s$ which is a small dictionary $\mathcal{D}_2^s$ sampled directly from $\mathcal{C}_2$, combined with $\mathcal{D}_1$. We report the compression ratio for each tranche, $\mathcal{R}(\mathcal{C}_2, \mathcal{D}_\cup^\bullet)$, and also for the whole collection in each round, $\mathcal{R}_{total}$. In every multi-round experiment, $\mathcal{C}_1$, $\mathcal{D}_1$, and so forth are *updated* for each new tranche. For example, when considering CuD, $\mathcal{D}_\cup^\Delta$ in round i becomes $\mathcal{D}_1$ in round $i+1$. Results for the CuD approach are in Table 1 and those for the baseline are in Table 2. The

[2]lemurproject.org/clueweb09

Table 4: Results of applying the CuD method for the updating scenario to five 50-GB tranches from the prefix of Wikipedia. The settings of threshold λ are the same as in Table 1.

Updating round	B (MB)	$\lvert\mathcal{C}_1\rvert$ (GB)	$\lvert\mathcal{D}_1\rvert$ (MB)	$\lvert\mathcal{C}_2\rvert$ (GB)	$\lvert\mathcal{C}_2^L\rvert$ (GB)	$\lvert\mathcal{D}_2^\Delta\rvert$ (MB)	$\lvert\mathcal{D}_\cup^\Delta\rvert$ (MB)	$\mathcal{R}(\mathcal{C}_2,\mathcal{D}_\cup^\Delta)$ (%)	$\lvert\mathcal{C}_1+\mathcal{C}_2\rvert$ (GB)	$\mathcal{R}(\mathcal{C}_1,\mathcal{D}_1)$ (%)	$\mathcal{R}_{total}$ (%)
0	1000	0	0	50	50.0	1000	1000	8.877	50	-	8.877
1	1250	50	1000	50	12.0	250	1250	**8.814**	100	8.877	8.846
2	1500	100	1250	50	11.8	250	1500	**8.390**	150	8.846	8.694
3	1750	150	1500	50	12.7	250	1750	**8.643**	200	8.694	8.681
4	2000	200	1750	50	11.3	250	2000	**7.781**	250	8.681	8.501

Table 5: Results of applying baseline method for the updating scenario to five 50-GB tranches from the prefix of Wikipedia. (Note that $\mathcal{C}_2^L$ here is exactly the same as $\mathcal{C}_2$, because $\mathcal{D}_2^s$ is sampled from the entire $\mathcal{C}_2$.)

Updating round	B (MB)	$\lvert\mathcal{C}_1\rvert$ (GB)	$\lvert\mathcal{D}_1\rvert$ (MB)	$\lvert\mathcal{C}_2\rvert$ (GB)	$\lvert\mathcal{C}_2^L\rvert$ (GB)	$\lvert\mathcal{D}_2^s\rvert$ (MB)	$\lvert\mathcal{D}_\cup^s\rvert$ (MB)	$\mathcal{R}(\mathcal{C}_2,\mathcal{D}_\cup^s)$ (%)	$\lvert\mathcal{C}_1+\mathcal{C}_2\rvert$ (GB)	$\mathcal{R}(\mathcal{C}_1,\mathcal{D}_1)$ (%)	$\mathcal{R}_{total}$ (%)
0	1000	0	0	50	50.0	1000	1000	8.877	50	-	8.877
1	1250	50	1000	50	50.0	250	1250	9.676	100	8.877	9.277
2	1500	100	1250	50	50.0	250	1500	9.373	150	9.277	9.309
3	1750	150	1500	50	50.0	250	1750	9.427	200	9.309	9.338
4	2000	200	1750	50	50.0	250	2000	8.833	250	9.338	9.237

threshold λ in CuD is set to *twice* the average factor length of $\mathcal{F}(\mathcal{C}_2,\mathcal{D}_1)$ (each tranche has its own λ).

The CuD method is more effective than the baseline in terms of both the compression ratio for each tranche and the compression of the whole collection. Moreover, compression improves as the threshold increases; briefly, if λ equals the average factor length, then the resulting $\mathcal{R}(\mathcal{C}_2,\mathcal{D}_\cup^\Delta)$ of Round 4 is 12.44 %, which drops to 12.18 % (result shown in Table 1) when λ is doubled.

For comparison, in Table 3, we also show the results of using only the initial (static) dictionary $\mathcal{D}^1$ to compress the incremental collections. These results confirm the need to tailor RLZ to the incremental corpus scenario.

To see how our scheme performs on a larger dataset, we divide the first 250 GB of the Wikipedia collection into five 50-GB tranches. Tables 4 and 5 together show an even greater win by CuD over the baseline. In particular, on the fifth tranche our compression ratio (7.781 %) is, relatively speaking, 11.9 % better than the baseline achieves (8.833 %).

To further demonstrate CuD's quality, we consider an adversarial experiment in which a dictionary of size B is sampled directly and only from tranche $\mathcal{C}_2$. Surprisingly, the results on the Wikipedia tranches, as shown in Table 6, are inferior to the results of CuD in Table 4.

In addition, recent work of Tong et al. [31] shows that in general a pruned dictionary compresses more effectively than a commensurate sampled dictionary. Thus we consider one further baseline, quite similar to that of Table 5. Here, however, the auxiliary dictionary is 'sampled and pruned' (with CARE) instead of just sampled. That is, for each update round, a 1000 MB $\mathcal{D}_2^s$ is sampled first from $\mathcal{C}_2$ and is then pruned to a 250 MB $\mathcal{D}_2^k$, which serves as the auxiliary dictionary. As shown in Table 7, the resulting tranche compression ratios ($\mathcal{R}(C_2,\mathcal{D}_\cup^k)$) are indeed significantly better than those in Table 5, though still inferior to CuD. In fact, should $\mathcal{D}_2^\Delta$ in Table 4 be generated from $\mathcal{C}_2^L$ first and then CARE-pruned, then even better performance could be achieved by CuD. To save space, we omit the details. Nevertheless, we observe that the competing methods in Table 6 and in Table 7 have an unfair advantage over CuD, as they

Table 6: Performance of compressing each tranche with a newly generated commensurate dictionary. $\mathcal{D}_2^o$ is completely generated from $\mathcal{C}_2$ and ignores $\mathcal{D}_1$. The Wikipedia tranches are the same as in Tables 4 and 5.

Round	B (MB)	$\lvert\mathcal{C}_2\rvert$ (GB)	$\lvert\mathcal{D}_2^o\rvert$ (MB)	$\mathcal{R}(\mathcal{C}_2,\mathcal{D}_2^o)$ (%)	$\mathcal{R}_{total}$ (%)
0	1000	50	1000	8.877	8.877
1	1250	50	1250	8.897	8.887
2	1500	50	1500	8.452	8.742
3	1750	50	1750	8.710	8.734
4	2000	50	2000	7.891	8.565

Table 7: Performance of compressing each tranche with an auxiliary dictionary that is a 1000 MB sample from $\mathcal{C}_2$, and then pruned to $\mathcal{D}_2^k$. The dataset used is the same as Table 5.

Round	B (MB)	$\lvert\mathcal{C}_2\rvert$ (GB)	$\lvert\mathcal{D}_2^k\rvert$ (MB)	$\lvert\mathcal{D}_\cup^k\rvert$ (MB)	$\mathcal{R}(C_2,\mathcal{D}_\cup^k)$ (%)	$\mathcal{R}_{total}$ (%)
0	1000	50	1000	1000	8.877	8.877
1	1250	50	250	1250	9.176	9.027
2	1500	50	250	1500	8.802	8.952
3	1750	50	250	1750	8.881	8.934
4	2000	50	250	2000	8.215	8.790

violate the memory constraint: $|\mathcal{D}_1|+|\mathcal{D}_\Delta| \leq B$. Even then, CuD outperforms each of them!

Finally, there is some evidence that the CuD algorithm compresses the Wikipedia collection about as well as RLZ with a static dictionary from the whole collection. With a static dictionary, the compression ratio for all 251 GB with a 2 GB dictionary is 8.688 %. CuD however compresses the initial 250 GB with an overall ratio of 8.501 %. Though the last 1 GB is missing from the latter experiment, this suggests CuD might inform better dictionary construction principles for the original RLZ algorithm. Furthermore, since CuD does not require recompression of previously compressed material, the retrieval speed of RLZ is unaffected.

4. INCREMENTAL COMPRESSION FOR REMOTE TRANSMISSION

We now turn to the second application of incremental compression, the transmitting scenario (Figure 1(b)). In some settings, incremental compression for remote transmission is quite similar to that for local archiving (Section 3). If the remote receiver has sufficient memory, that is $B >> |\mathcal{D}_1|$, then we can simply adopt the CuD approach. However, we might already have saturated the available memory, $B = |\mathcal{D}_1|$, and then even a tiny $\mathcal{D}_\Delta$ cannot be added to $\mathcal{D}_1$; in this section, we offer a new approach. We still generate a compact auxiliary dictionary $\mathcal{D}_\Delta$, but this time we do not combine it with all of $\mathcal{D}_1$. Rather, we retain only the 'useful' parts of $\mathcal{D}_1$ when processing $\mathcal{C}_2$. In this context, we formulate the problem as:

> Given a tranche $\mathcal{C}_2$, an existing dictionary $\mathcal{D}_1$, and also a memory budget B, extract a smaller dictionary $\mathcal{D}_1'$ from $\mathcal{D}_1$ and generate an auxiliary dictionary $\mathcal{D}_\Delta$ from $\mathcal{C}_2$ to minimize
>
> $$|\mathcal{F}(\mathcal{C}_2, \mathcal{D}_1' + \mathcal{D}_\Delta)|_c + |\mathcal{D}_\Delta|_c + |\mathcal{D}_1'|_c\,,$$
>
> while guaranteeing that $|\mathcal{D}_1'| + |\mathcal{D}_\Delta| \le B$.

Here, $|\mathcal{D}_\Delta|_c$ and $|\mathcal{D}_1'|_c$ are the sizes of compact representations of $\mathcal{D}_\Delta$ and $\mathcal{D}_1'$, respectively (there is a glossary in Table A.2).

To retrieve files quickly, uncompressed dictionaries $\mathcal{D}_\Delta$ and $\mathcal{D}_1'$ must reside in RAM. Hence the memory constraint cannot be $|\mathcal{D}_1'|_c + |\mathcal{D}_\Delta|_c \le B$. During transmission, however, the dictionaries should be compressed. Again the size of the ancillary data structure for string search, for example a suffix array, is proportional to the dictionary size, so we ignore this constant-factor overhead and focus solely on dictionary size. Moreover, this ancillary structure can be derived from the dictionary, and need not be transmitted. To simplify the presentation, in the rest of this section, we assume that $B = |\mathcal{D}_1|$.

4.1 Interrogating the existing dictionary

The two obvious baselines for processing $\mathcal{C}_2$ are (i) $\mathcal{D}_1' = \mathcal{D}_1$ with $\mathcal{D}_\Delta = \emptyset$, and (ii) $\mathcal{D}_1' = \emptyset$ with $\mathcal{D}_\Delta = \mathcal{D}_2^s$, where $\mathcal{D}_2^s$ is sampled from $\mathcal{C}_2$. However, the results in the previous section show that such extreme choices are unlikely to yield optimal performance. That is, each of the original dictionaries ($\mathcal{D}_1$ and $\mathcal{D}_2^s$) is valuable when compressing $\mathcal{C}_2$.

CnP (Concatenating and Pruning). Motivated by the findings in previous work [31] that effective pruning helps to retain the most 'valuable' parts of the dictionary, we introduce the CnP scheme to generate both $\mathcal{D}_1'$ and $\mathcal{D}_\Delta$ (Algorithm 2). First, a relatively large dictionary $\mathcal{D}_2^s$ is sampled from tranche $\mathcal{C}_2$ and is concatenated to the existing dictionary $\mathcal{D}_1$ to form a even larger dictionary $\mathcal{D}'$. To simplify the presentation in the rest of this section, we assume $|\mathcal{D}_2^s| = B$, though of course it is the final dictionary, $\mathcal{D}_2$, that *must* fit in the bound B. We then apply a principled pruning method to reduce $\mathcal{D}'$ to $\mathcal{D}_2$, whose size is B. In Algorithm 2, `Prune()` in line 3 corresponds to the CARE pruning approach [31]. Dictionary $\mathcal{D}_2$ as a whole is the RAM-resident dictionary for the new tranche $\mathcal{C}_2$. However, to update efficiently the remote server with the new tranche, we must describe the dictionary in a compressed form, and must thus consider the two constituent components of $\mathcal{D}_2$: $\mathcal{D}_1'$ (from $\mathcal{D}_1$) and $\mathcal{D}_\Delta$ (from D_2^s), represented as $\mathcal{D}_1^p$ and $\mathcal{D}_2^p$ in Figure 6, respectively.

Algorithm 2 The CnP approach for incremental compression in the transmitting scenario.

```
Input: Incremental collection $\mathcal{C}_2$, Existing dictionary $\mathcal{D}_1$,
    Memory budget size $B$ (we assume $|\mathcal{D}_1| = B$)
1: $\mathcal{D}_2^s \leftarrow$ Sample($\mathcal{C}_2, B$)
2: $\mathcal{D}' \leftarrow \mathcal{D}_1 + \mathcal{D}_2^s$
3: $\mathcal{D}_2 \leftarrow$ Prune($\mathcal{C}_2, \mathcal{D}', B$)
4: $\mathcal{D}_1^p \leftarrow \mathcal{D}_1$-derived parts in $\mathcal{D}_2$
5: $\mathcal{D}_2^p \leftarrow \mathcal{D}_2 - \mathcal{D}_1^p$
6: $\mathcal{F}(\mathcal{C}_2, \mathcal{D}_2) \leftarrow$ Factorize($\mathcal{C}_2, \mathcal{D}_2$)
7: Transmit compressed $\mathcal{F}(\mathcal{C}_2, \mathcal{D}_2)$ and compact representations for $\mathcal{D}_1^p$ and $\mathcal{D}_2^p$
```

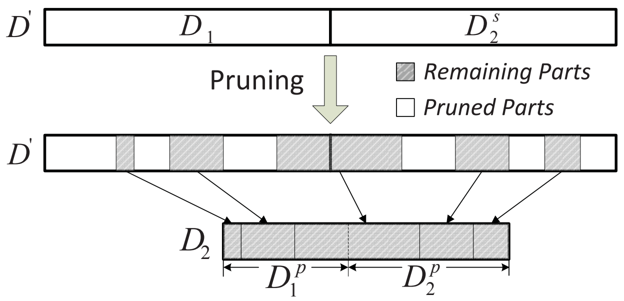


Figure 6: Obtaining $\mathcal{D}_1'$ and $\mathcal{D}_\Delta$ (denoted by D_1^p and D_2^p in the diagram).

4.2 Encoding the fragments of the dictionary

Suppose we have already derived a dictionary $\mathcal{D}_2$ for tranche $\mathcal{C}_2$; and suppose this $\mathcal{D}_2$ contains some strings from the existing dictionary (denoted by $\mathcal{D}_1^p$) and some content drawn from the new tranche (denoted by $\mathcal{D}_2^p$). The remote server should construct $\mathcal{D}_2$ from a description of each of $\mathcal{D}_1^p$ and $\mathcal{D}_2^p$. By design, $\mathcal{D}_2^p$ ought to comprise material that the remote server does not yet have, and thus this material must be transmitted.

However, sending $\mathcal{D}_1^p$ in its entirety, even in a compressed form, is unnecessary. Given that the receiver already has a copy of $\mathcal{D}_1$, we save bandwidth by describing $\mathcal{D}_1^p$ in terms of the substrings of $\mathcal{D}_1$ that it represents. (We assume here that $\mathcal{D}_1^p$ is concatenated with $\mathcal{D}_2^p$, thence no information is required to describe the relative locations of these sub-dictionaries.)

Consider the example in Figure 6. Dictionary $\mathcal{D}_1^p$ is derived from three runs of bytes in $\mathcal{D}_1$; this 'included' material is shown in gray. There are also three white runs of 'excluded' or pruned material. An obvious way to represent these is as a sequence of pairs: each pair consists of the position of the start of each gray run together with its length. The positions and lengths then need to be encoded. Although in general a Golomb code is suitable for compressing a sequence of positions in an array of known size, in this case we have additional information (factor lengths) that could be incorporated into the compression, but are ignored.

An alternative is to regard the sequence of gray runs as a sequence of lengths, of known total length, and the sequence of white runs as, likewise, a sequence of lengths of known total length. We assuming that whether each posi-

Table 8: Results of applying the CnP and baseline methods for the transmitting scenario on different small datasets*. Dictionary sizes are $|\mathcal{D}_1| = |\mathcal{D}_2^s| = 1000$ MB. $|\mathcal{D}_2^p|_c$ here denotes the size of 7zip-compressed $\mathcal{D}_2^p$ size, while $|\mathcal{D}_1^p|_c$ represents the size of Golomb-encoded run-length information for recovering $\mathcal{D}_1^p$.

Dataset Types	$\mathcal{C}_1$	$\mathcal{C}_2$	Strategies for constructing $\mathcal{D}_2$ ($\|\mathcal{D}_2\|$=1000 MB)	Ratio (%) $\mathcal{R}(\mathcal{C}_2, \mathcal{D}_2)$	$\|\mathcal{D}_2^p\|_c/\|\mathcal{C}_2\|$	$\|\mathcal{D}_1^p\|_c/\|\mathcal{C}_2\|$	Transmit Ratio (%)
Homogeneous	G_1	G_2	*only use* $\mathcal{D}_1$	17.16	-	-	17.16
			only use $\mathcal{D}_2^s$	12.97	0.58	-	13.55
			CnP (*use both* $\mathcal{D}_1$ *and* $\mathcal{D}_2^s$)	12.45	0.48	0.020	12.97
Heterogeneous	G_1	W_1	*only use* $\mathcal{D}_1$	36.81	-	-	36.81
			only use $\mathcal{D}_2^s$	7.99	0.45	-	8.44
			CnP (*use both* $\mathcal{D}_1$ *and* $\mathcal{D}_2^s$)	7.89	0.42	0.004	8.31

* G_1/G_2: the first/second 25 GB of GOV2; W_1/W_2: the first/second 25 GB of Wikipedia.

Table 9: Results of applying the CnP and baseline methods for the transmitting scenario on different large datasets*. Dictionary sizes are $|\mathcal{D}_1| = |\mathcal{D}_2^s| = 1000$ MB. The meanings of $|\mathcal{D}_1^p|_c$ and $|\mathcal{D}_2^p|_c$ are the same as Table 8.

Dataset Types	$\mathcal{C}_1$	$\mathcal{C}_2$	Strategies for constructing $\mathcal{D}_2$ $\|\mathcal{D}_2\|$=1000 MB	Ratio (%) $\mathcal{R}(\mathcal{C}_2, \mathcal{D}_2)$	$\|\mathcal{D}_2^p\|_c/\|\mathcal{C}_2\|$	$\|\mathcal{D}_1^p\|_c/\|\mathcal{C}_2\|$	Transmit Ratio (%)
Homogeneous Datasets	W_1	W_2	*only use* $\mathcal{D}_1$	11.32	-	-	11.32
			only use $\mathcal{D}_2^s$	9.82	0.12	-	9.94
			CnP (*use both* $\mathcal{D}_1$ *and* $\mathcal{D}_2^s$)	8.96	0.09	0.003	9.05
	G_1	G_2	*only use* $\mathcal{D}_1$	6.96	-	-	6.96
			only use $\mathcal{D}_2^s$	5.43	0.07	-	5.50
			CnP (*use both* $\mathcal{D}_1$ *and* $\mathcal{D}_2^s$)	4.99	0.05	0.001	5.04
	G_2	G_1	*only use* $\mathcal{D}_1$	18.60	-	-	18.60
			only use $\mathcal{D}_2^s$	12.96	0.14	-	13.10
			CnP (*use both* $\mathcal{D}_1$ *and* $\mathcal{D}_2^s$)	12.63	0.13	0.002	12.76
Heterogeneous Datasets	W_1	G_1	*only use* $\mathcal{D}_1$	39.94	-	-	39.94
			only use $\mathcal{D}_2^s$	12.96	0.14	-	13.10
			CnP (*use both* $\mathcal{D}_1$ *and* $\mathcal{D}_2^s$)	12.86	0.14	0.002	13.00
	G_1	W_2	*only use* $\mathcal{D}_1$	37.64	-	-	37.64
			only use $\mathcal{D}_2^s$	9.82	0.12	-	9.93
			CnP (*use both* $\mathcal{D}_1$ *and* $\mathcal{D}_2^s$)	9.68	0.11	0.002	9.79

* W_1/W_2: the first/second 100 GB of Wikipedia; G_1/G_2: the first/second 100 GB of GOV2.

tion is the start of a run—in a concatenation of gray (or white) runs—is an observation from an independent but identical, Bernoulli trials. A sequence of a known number of run lengths of known total length almost forms a geometric distribution: this can be bitwise optimally represented by a Golomb code [12, 22].

To confirm that a Golomb code is appropriate, we examined 126 points from the empirical cumulative distribution of the run lengths generated by two tranches of GOV2. We plotted these quantiles against quantiles from a geometric distribution whose parameter was derived from the observed mean run length. This Q-Q plot is very close to linear, and has correlation 0.995.

Moreover, in our experiments, we found that this approach yields a compact code, requiring for example only 3.2 MB to describe about 2 million runs of gray in a 1000 MB dictionary; this was far less than was required by other methods we explored.

4.3 Experimental results

As we do in the updating scenario, in this (transmitting) scenario, we evaluate our algorithm (here, CnP) on both small and large datasets. We take two consecutive tranches from the prefix of GOV2, called G_1 and G_2, and two from the prefix of the Wikipedia dataset, called W_1 and W_2. In our first experiment, reported in Table 8, each tranche is 25 GB; in our second experiment, reported in Table 9, each tranche is 100 GB. In each experiment, we investigate both a *homogeneous* setup, with the two tranches from the same source, and a *heterogeneous* setup, with two tranches from different sources.

Since our goal is to minimize the total data transmitted we outline a new metric, the Transmit Ratio (TR). The TR consists of three parts, each of which is reported as a ratio to the uncompressed size of the tranche $\mathcal{C}_2$:

1. $\mathcal{R}(\mathcal{C}_2, \mathcal{D}_2)$: The compression ratio of $\mathcal{C}_2$.

2. $|\mathcal{D}_2^p|_c/|\mathcal{C}_2|$: The size ratio of an efficient representation of $\mathcal{D}_2^p$ to the uncompressed $\mathcal{C}_2$. Here we use `7zip` to compress $\mathcal{D}_2^p$ before transmitting it.

3. $|\mathcal{D}_1^p|_c/|\mathcal{C}_2|$: The size ratio of an efficient representation of $\mathcal{D}_1^p$ to the uncompressed $\mathcal{C}_2$. As discussed above, we describe $\mathcal{D}_1^p$ as two sequences of *locations* in $\mathcal{D}_1$ and

thence Golomb encode each sequence. The size is zero (represented by -) if either all or none of $\mathcal{D}_1$ is used.

As we can see in Tables 8 and 9, CnP, which takes advantage of both $\mathcal{D}_1$ and $\mathcal{D}_2^s$ during constructing $\mathcal{D}_2$, outperforms each of the two extreme baselines. The gap in performance is especially apparent with homogeneous datasets, and for heterogeneous datasets, the baseline that only uses $\mathcal{D}_1$ performs extremely badly. Besides, we can also observe that the Golomb code does a good job in representing $\mathcal{D}_1^p$.

As an example, on the large Wikipedia tranches, CnP achieves a Transmit Ratio of 9.054 %, relatively speaking, this is 8.9 % better than the baseline dictionary derived solely from $\mathcal{C}_2$. Thus, we regard the two strategies (CnP and CuD) together as a comprehensive solution to the incremental compression problem.

5. CONCLUSIONS

Relative Lempel-Ziv factorization is a fast and effective compression scheme for large repositories. As initially designed, it maintains a static dictionary in RAM against which a static collection can be efficiently encoded and decoded. Previous work has explored compressing new material with a static dictionary, but in some cases, as confirmed in our experiments, this leads to poor outcomes. A naïve solution is to generate a new dictionary solely from the new tranche, thus adding a dictionary with each tranche, which is unlikely to be sustainable.

We have described new methods for RLZ compression of new material based on relatively small extensions to the original dictionary. Our work considers two practical incremental collection scenarios: archiving and transmitting large incremental repositories while maintaining fast retrieval and reasonable RAM requirements; in these scenarios, the collection is extended by addition of tranches of new material. We exploit the observation that it is likely that a significant portion of a dictionary derived from a new tranche shares many common strings with a dictionary derived from existing tranches. We therefore describe a scheme that successfully generates an effective and compact auxiliary dictionary, based on consideration of the existing dictionary.

We focus on methods that require no recompression of previously compressed material, and have no impact on the extremely impressive decompression speed that can be achieved by RLZ. We concede that the compression times of CuD and CnP are larger than the baseline, but only because they each invoke RLZ twice: the fundamental principles of RLZ are unchanged.

We show that such a combined new dictionary can achieve much better compression ratio (almost 12 % relatively speaking) than if only the previous dictionary is used, and helps avoid pathological cases that arise when the character of the data is changed. We have also shown that representation of a large number of substrings of the original dictionary can be highly compact, through application of Golomb codes. For a comparable compression ratio, our combined new dictionary requires much less memory than that required by the naïve baseline. If the newly generated dictionaries are commensurate in size, our method dramatically outperforms the naïve solution. Overall, we show that new material can be added to a repository with only limited cost.

6. ACKNOWLEDGMENTS

This work is partially supported by The Australian Research Council, NSF of China (61373018, 11301288), Program for New Century Excellent Talents in University (NCET-13-0301) and Fundamental Research Funds for the Central Universities (65141021). Jiancong Tong would also like to thank The China Scholarship Council (CSC) for the State Scholarship Fund.

7. REFERENCES

[1] R. A. Baeza-Yates and B. A. Ribeiro-Neto. *Modern Information Retrieval—the concepts and technology behind search*. Addison-Wesley, second edition, 2011.

[2] J. L. Bentley and M. D. McIlroy. Data compression with long repeated strings. *Information Sciences*, 135(1–2):1–11, 2001.

[3] N. R. Brisaboa, A. Fariña, G. Navarro, and J. R. Paramá. Compressing dynamic text collections via phrase-based coding. In *ECDL*, pages 462–474, 2005.

[4] S. Büttcher, C. L. A. Clarke, and G. V. Cormack. *Information Retrieval—Implementing and Evaluating Search Engines*. MIT Press, 2010.

[5] A. Cannane and H. E. Williams. A general-purpose compression scheme for large collections. *ACM Transactions on Information Systems*, 20(3):329–355, 2002.

[6] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Transaction on Computing System*, 26(2):4:1–4:26, 2008.

[7] C. Clarke, N. Craswell, and I. Soboroff. Overview of the TREC 2004 terabyte track. In *TREC*, 2004.

[8] W. B. Croft, D. Metzler, and T. Strohman. *Search Engines—Information Retrieval in Practice*. Addison-Wesley, 2009.

[9] P. Ferragina and G. Manzini. On compressing the textual web. In *WSDM*, pages 391–400, 2010.

[10] P. Ferragina and R. Venturini. The compressed permuterm index. *ACM Transactions on Algorithms*, 7(1):10, 2010.

[11] G. Francès, X. Bai, B. B. Cambazoglu, and R. A. Baeza-Yates. Improving the efficiency of multi-site web search engines. In *WSDM*, pages 3–12, 2014.

[12] S. W. Golomb. Run-length encodings. *IEEE Transactions on Information Theory*, 12(3):399–401, 1966.

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

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

[15] 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.

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

[17] J. J. Hunt, K.-P. Vo, and W. F. Tichy. Delta algorithms: An empirical analysis. *ACM Transactions on Software Engineering and Methodology*, 7(2):192–214, 1998.

[18] P. Kulkarni, F. Douglis, J. D. LaVoie, and J. M. Tracey. Redundancy elimination within large collections of files. In *USENIX Annual Technical Conference, General Track*, pages 59–72, 2004.

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

[20] U. Manber and E. W. Myers. Suffix arrays: A new method for on-line string searches. *SIAM Journal on Computing*, 22(5):935–948, 1993.

[21] C. D. Manning, P. Raghavan, and H. Schütze. *Introduction to information retrieval*. Cambridge University Press, 2008.

[22] A. Moffat and A. Turpin. *Compression and Coding Algorithms*. Kluwer, 2002.

[23] A. Moffat, J. Zobel, and N. Sharman. Text compression for dynamic document databases. *IEEE Trans. Knowl. Data Eng.*, 9(2):302–313, 1997.

[24] G. Navarro and V. Mäkinen. Compressed full-text indexes. *ACM Computing Surveys*, 39(1), 2007.

[25] Z. Ouyang, N. D. Memon, T. Suel, and D. Trendafilov. Cluster-based delta compression of a collection of files. In *WISE*, pages 257–268, 2002.

[26] A. Peel, A. Wirth, and J. Zobel. Collection-based compression using discovered long matching strings. In *CIKM*, pages 2361–2364, 2011.

[27] P. Shilane, M. Huang, G. Wallace, and W. Hsu. WAN-optimized replication of backup datasets using stream-informed delta compression. *ACM Transations on Storage*, 8(4):1–26, 2012.

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

[29] T. Suel, P. Noel, and D. Trendafilov. Improved file synchronization techniques for maintaining large replicated collections over slow networks. In *ICDE*, pages 153–164, 2004.

[30] D. Teodosiu, N. Bjørner, Y. Gurevich, M. Manasse, and J. Porkka. Optimizing file replication over limited-bandwidth networks using remote differential compression. Technical Report TR2006-157-1, Microsoft Research, 2006.

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

[32] H. E. Williams and J. Zobel. Compressing integers for fast file access. *The Computer Journal*, 42(3):193–201, 1999.

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

[34] J. Zhang, X. Long, and T. Suel. Performance of compressed inverted list caching in search engines. In *WWW*, pages 387–396, 2008.

[35] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977.

[36] N. Ziviani, E. Silva de Moura, G. Navarro, and R. A. Baeza-Yates. Compression: A key for next-generation text retrieval systems. *IEEE Computer*, 33(11):37–44, 2000.

[37] J. Zobel and A. Moffat. Inverted files for text search engines. *ACM Computing Surveys*, 38(2), 2006.

APPENDIX

A. GLOSSARY

Table A.1: Notation in Section 3.

B	Budget for RAM-resident dictionary
$\mathcal{F}(\mathcal{C},\mathcal{D})$	Factorization of (collection) $\mathcal{C}$ against (dictionary) $\mathcal{D}$: a sequence of factors
$\mathcal{R}(\mathcal{C},\mathcal{D})$	The corresponding compression ratio of RLZ for (collection) $\mathcal{C}$ and (dictionary) $\mathcal{D}$
$\mathcal{C}_1,\mathcal{C}_2$	The existing and new tranches
$\mathcal{D}_1,\mathcal{D}_2$	Dictionaries, from $\mathcal{C}_1$ and $\mathcal{C}_2$ (respectively)
$\mathcal{D}_\Delta$	Auxiliary dictionary for $\mathcal{C}_2$
$\mathcal{D}_\cup$	Concatenation of $\mathcal{D}_1$ and $\mathcal{D}_\Delta$
$\mathcal{D}'$	A dictionary sampled directly from $\mathcal{C}_2$, which will later be reduced to $\mathcal{D}_\Delta$
$\mathcal{D}_2^s$	A dictionary sampled directly from $\mathcal{C}_2$
$\mathcal{D}_2^k$	Dictionary CARE pruned from $\mathcal{D}_2^s$
$\mathcal{D}_2^\Delta$	Auxiliary dictionary that is the outcome of the CuD algorithm
$\mathcal{D}_\cup^\bullet$	Concatenation of $\mathcal{D}_1$ and $\mathcal{D}_2^\bullet$ ($\bullet$ can either be Δ, s, or k)
λ	Length threshold for identifying 'small' factors
$\mathcal{F}_\lambda$	Sequence of all 'small' factors from C_2
$\mathcal{C}_2^L$	Concatenation of all 'small' factors from C_2 that are picked by the CuD algorithm
$\mathcal{D}_2'$	Dictionary sampled from $\mathcal{C}_2^L$ (for Figure 5)
$\mathcal{D}^1$	Dictionary from the *initial* tranche (baseline in Table 3)
$\mathcal{D}_2^o$	Dictionary generated from $\mathcal{C}_2^o$, which is large enough and will not be concatenated with $\mathcal{D}_1$ (baseline in Table 6)

Table A.2: Notation in Section 4.

$\lvert\mathcal{F}(\mathcal{C},\mathcal{D})\rvert_c$	The compressed size of $\mathcal{F}(\mathcal{C},\mathcal{D})$ (using `ZLIB`/`VBYTE` for encoding positions/lengths)
$\mathcal{D}_1'$	Smaller dictionary extracted from $\mathcal{D}_1$
$\lvert\mathcal{D}_1'\rvert_c$	The size of Golomb-encoded representation of the parts of $\mathcal{D}$ that are in $\mathcal{D}_1'$
$\lvert\mathcal{D}_\Delta\rvert_c$	The size of 7-zip-compressed $\mathcal{D}_\Delta$
$\mathcal{D}'$	Concatenation of $\mathcal{D}_1$ and $\mathcal{D}_\Delta$
$\mathcal{D}_1^p$	CnP-pruned $\mathcal{D}_1'$ from $\mathcal{D}_1$
$\mathcal{D}_2^p$	CnP-pruned $\mathcal{D}_\Delta'$ from $\mathcal{D}_2^s$
G_1, G_2	The first and second tranche of GOV2
W_1, W_2	The first and second tranche of Wikipedia