A Cost Model for Long-Term Compressed Data Retention

Kewen Liao[1] Alistair Moffat Matthias Petri Anthony Wirth
Department of Computing and Information Systems
The University of Melbourne
Melbourne, Australia

ABSTRACT

Vast amounts of data are collected and stored every day, as part of corporate knowledge bases and as a response to legislative compliance requirements. To reduce the cost of retaining such data, compression tools are often applied. But simply seeking the best compression ratio is not necessarily the most economical choice, and other factors also come in to play, including compression and decompression throughput, the main memory required to support a given level of on-going access to the stored data, and the types of storage available. Here we develop a model for the total retention cost (TRC) of a data archiving regime, and by applying the charging rates associated with a cloud computing provider, are able to derive dollar amounts for a range of compression options, and hence guide the development of new approaches that are more cost-effective than current mechanisms. In particular, we describe an enhancement to the Relative Lempel Ziv (RLZ) compression scheme, and show that in terms of TRC, it outperforms previous approaches in terms of providing economical long-term data retention.

1. INTRODUCTION

Long-term data retention is a substantial cost for all corporate and government entities. For example, Internet service providers are required to store customer email and web access metadata for periods measured in years, and financial transaction records and call-center voice recordings might be retained for a decade or more. Indeed, the stored data associated with query logs, click logs, and crawled documents is a key corporate asset of web services companies such as Bing or Google. Because of the data volumes involved, compression techniques (including data deduplication, see, for example, Kulkarni et al. [8]) are widely used, to reduce the storage space needed, and hence the cost. For example, the ZLib library[2] and the gzip tool based on it have been widely used for several decades; and the more recent bzip2[3] and xz[4] approaches provide even better compression. But choosing between compression techniques is more complex than just applying the one that provides the best compression ratio. Other factors that must be considered include the throughput rates at which data can be compressed and decompressed; the frequency with which the compressed data will be accessed during the retention period; the query latency within which such requests must be responded to; the amount of memory space required during compression and decompression; the minimum time between when data is received by the storage system and when it must be available for access; secondary storage costs relative to main memory costs; and so on.

Here we argue that the *total retention cost* (TRC) can be reliably modeled and used as an all-embracing measure of a compressed storage regime, with all of the various components mapped through to dollars, at rates that reflect the particular mix of technologies involved. We add to this the widely-used approach of a *service-level agreement* (SLA), in this case a statement of requirements as to the volume of data to be added to the archive in each corresponding time period; the minimum duration for which the data must be held; the number of access requests that must be supported; and the maximum latency within which any particular access request must be responded to. The goal of the system designer is then to determine the compression tool and storage arrangement that has the lowest TRC while still complying with the SLA.

By profiling different compression programs, and making use of hardware costings provided by a major cloud computing provider, we are able to model and compare the dollar cost of a range of data storage scenarios, and determine the critical relationships that influence tool selection. The cost model has also suggested new combinations, and we describe a refinement to the Relative Lempel Ziv (RLZ) method [7, 10] that results in reduced data retention costs.

Contributions. Our work provides two complementary ideas:

- A detailed cost model for compressed data retention, in which encoding and decoding costs are properly weighed against compression effectiveness, and alternative mechanisms can be compared in economic terms; and
- An improved RLZ compression scheme that responds to the various tensions embedded in the cost model, and provides data retention that is demonstrably more cost-effective than previous methods.

The remainder of the paper addresses these complementary claims.

2. COMPRESSION OPTIONS

We briefly survey compression options. For a more comprehensive treatment, see Witten et al. [15] or Moffat and Turpin [11]. We

[1] Current address: Department of Computer Science and Software Engineering, Swinburne University of Technology, Melbourne, Australia.

[2] https://github.com/madler/zlib.

[3] http://www.bzip.org.

[4] http://tukaani.org/xz/.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

WSDM 2017, February 06–10, 2017, Cambridge, United Kingdom

© 2017 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ISBN 978-1-4503-4675-7/17/02...$15.00

DOI: http://dx.doi.org/10.1145/3018661.3018738

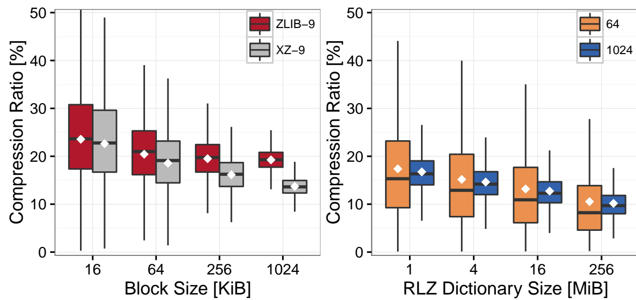

Figure 1: Adaptive and semi-static compression: ZLib and xz applied to a bale of 64 GiB of web data using a range of block sizes (left); and RLZ with 64 kiB and 1 MiB blocks applied using a range of dictionary sizes (right), including the cost of the compressed dictionary. The central dot plotted in each box is the average; the central line is the median. Full details of these two experiments and the data used are given in Section 6.

assume throughout that a *bale* of data, perhaps 1 GiB to 1 TiB, is to be represented as a sequence of compressed *blocks*, each containing some fixed volume of the incoming data, perhaps 4 kiB to 1 MiB. We further assume that the location of each compressed block within the bale is maintained in an index, and that each block is to be able to be decompressed independently of other blocks. We discuss bales and blocks more carefully in Section 3.

One key way of categorizing compression tools is whether they are *adaptive* or *semi-static*. Adaptive mechanisms process data in a single pass, dealing with the first byte of each block in a bland initial state, and building an evolving model of the block as it is compressed. The LZ77-type approach that is embedded in ZLib operates in this way (see Witten et al. [15] for more information). In contrast, semi-static mechanisms make use of an external model – often a stored *dictionary* – that has been prepared for that bale via a preliminary process, and is shared without further change while each of the blocks comprising the bale is processed. In both adaptive and semi-static models, variable-length strings (*factors*) within the block are identified with reference to the model, and represented in the compressed stream via fixed-length or variable-length codewords. *Access* requests, to extract a sequence of bytes starting at a specified offset within the bale, are achieved by first using the bale's block index to determine the necessary block and a corresponding byte offset within that block. The compressed form of the block is then retrieved from secondary storage, and decoded in entirety from its beginning so that the required byte stream can be returned.

The disadvantage of adaptive approaches is that compression effectiveness (also referred to as the compression *ratio*) is relatively poor for short blocks, during the period through which the model is still accumulating statistics about the nature of the data being represented. This relationship suggests that blocks should be as large as possible. The left pane of Figure 1 shows how compression ratio (expressed as a percentage relative to the uncompressed size) for the standard tools ZLib and xz varies with the blocksize averaged over a 64 GiB bale drawn from the HGOV2 test data described in Section 6. Small blocks yield inferior compression compared to large blocks; and with xz in particular, compression continues to improve even after hundreds of kilobytes of data have been processed.

On the other hand, semi-static approaches achieve consistent compression effectiveness even for small blocks, because the dictionary is formed for the whole bale and then made available in full as each of the blocks is compressed, without there being any "learning" period at the start of the block. Semi-static methods have the disadvantage of requiring that the dictionary be somehow computed and stored along with the compressed data, and then held in memory during decoding. For example, the *Relative Lempel Ziv* (RLZ) mechanism of Hoobin et al. [7] extracts *segments* of 1 kiB from the bale at uniform intervals, and concatenates them to form a dictionary. The compressed form of a block is a list of greedily-selected maximal-length factors identified by their *offsets* and *lengths* in the dictionary, stored as coded integers; with *literals* used if a factor greater than some minimal length is not available [12]. Targeted segment selection processes have been shown to lead to improved compression effectiveness [10]. The *compression effectiveness* for RLZ is then the combined cost of separately representing the offsets, lengths, and literals. In the RLZ-ZZZ variant, all three integer streams are passed to ZLib to be represented [12]. As well, the further cost of storing the dictionary in compressed form must be included. High compression effectiveness arises when the dictionary is 0.5–1.0% of the text being compressed [7, 12]. For a bale of (say) 64 GiB, a dictionary of 256 MiB or more is thus suggested.

If the RLZ-compressed bale is to be online queryable, the dictionary must be held uncompressed in memory for the entire data retention period, a potentially onerous overhead given that main memory might cost 10 or 100 times as much per byte as does secondary storage. The right pane in Figure 1 shows how the RLZ-ZZZ compression ratio varies as dictionary size changes, using the same test bale as is shown in the left pane's results, and using two different blocksizes (again, see Section 6 for details). Compression effectiveness notably superior to ZLib and xz is possible if non-trivial memory can be allocated as the dictionary. Note that while compression ratios for RLZ also vary somewhat with blocksize because of the use of ZLib as a backend coder, the lengths of the factorizations developed are primarily dependent on the dictionary size only, and not on the blocksize. With the larger 1 MiB blocks there is more consistency in the compressed sizes, and the average effectiveness is slightly superior. A 16 MiB dictionary suffices with 64 kiB blocks to obtain better compression effectiveness than xz with 1 MiB blocks.

Another trade-off arises in terms of computational cost. If one option yields better compression effectiveness than another but takes longer to compute, the second one might have a lower TRC. The greater the data retention period, the smaller the compression effectiveness advantage needs to be before – all other things being equal – a higher initial compression cost can be justified.

3. A DATA RETENTION COST MODEL

We now describe the SLA and TRC model we propose, before calculating the cost of different approaches in the next section.

Service Level Agreement. We assume that data arrives at the retention service in uncompressed or simple-but-fast compressed format from a client, and must be ingested at a rate of λ GiB per day, perhaps as small as 1 GiB per day, or as large as 1 TiB per day or more. We also assume that incoming data is accumulated into *bales* of B GiB each, processed into queryable form, including being partitioned into equal-sized blocks and then compressed, and then *published* in order to meet a required minimum turnaround time L_W between ingest and availability for querying, for example $L_W = 1$ hour or $L_W = 1$ day. Once ingested, data must be *retained* for some minimum further period L_D, perhaps $L_D = 366$ days or $L_D = 7$ years. We further assume that access requests arrive at a rate of q queries per day per stored GiB (including possibly $q = 0$) throughout the retention period, and that each query must be responded to (by supplying the requested decompressed byte subsequence from within the bale) within a time limit of L_R, perhaps $L_R = 0.1$ or $L_R = 1$

Symbol	Unit	Description
SLA parameters		
λ	GiB/d	Data ingest rate
L_W	days	Maximum delay before publication
L_D	days	Minimum retention time
L_R	secs	Maximum latency for queries
q	#/GiB/d	Query arrival rate
Hardware parameters		
C_{io}	$/GiB	Cost of data transfer to/from disk
C_{disk}	$/TiB/y	Disk storage cost
C_{cpu}	$/PiCyc	Cost for CPU operations
C_{mem}	$/GiB/y	Main memory storage cost
Algorithm-specific parameters		
enc_{mem}	MiB	Memory space required during encoding
enc_{ini}	MiCyc	Encoding CPU fixed cost
enc_{spd}	Cyc/B	Encoding CPU variable cost
dec_{mem}	MiB	Memory space required during decoding
dec_{ini}	MiCyc	Decoding CPU fixed cost
dec_{spd}	Cyc/B	Decoding CPU variable cots
$ratio(b)$	%	Compression ratio for blocks of b bytes
Derived/computed parameters		
B	GiB	Bale size
b	kiB	Block size for compression and retrieval
$TRC(B)$	$/bale	Total retention cost for a bale of size B

Table 1: Glossary of symbols, with units of "B" representing bytes, "Cyc" representing CPU clockcycles, "d" representing days, "y" representing years, and the prefixes "ki", "Mi", "Gi" and "Pi" indicating 2^{10}, 2^{20}, 2^{30}, and 2^{40}, respectively.

seconds. A compression regime and hardware combination that satisfies the requirements of the data owner by virtue of meeting the terms of an SLA is said to be *compliant* with regard to the SLA.

The first section of Table 1 lists these variables and the typical units used to represent them. To compute costs from these quantities, conversions to other units may be required. For example, an ingest rate of $\lambda = 1\,\text{GiB/day} \approx 12.1\,\text{kiB/second}$, and a data retention time of 1 year $\approx$ 30.0 Mebisecond.

Hardware Support. The second section of Table 1 lists the hardware characteristics that determine the cost of compliance. The secondary storage employed is presumed to have a transfer cost of C_{io} dollars per GiB, and a storage cost of C_{disk} dollars per GiB per year. Similarly, the processor employed is presumed to cost C_{cpu} dollars per PebiCycle (PiCyc), and to be equipped with main memory that costs C_{mem} dollars per GiB per year. Clock cycles are an imperfect measure of the rate at which a CPU is able to execute a program, because memory stalls, variable CPU clock rates during execution and branch/cache misses also play a critical part; but even so, they are a good first approximation of relative CPU cost and speed for modern pipelined architectures, and we use the two terms interchangeably. In Section 4 we measure the correlation between clock cycles and program execution time.

Algorithm Characteristics. The third section of Table 1 lists a number of algorithm-dependent characteristics as they pertain to handling a block of b bytes of data, including the cost of encoding and decoding, both measured as clock cycles per (uncompressed) byte; and the encoding and decoding flagfall cost per block, possibly zero. The memory required during encoding and decoding also affects the computed cost, and is shown as enc_{mem} and dec_{mem} respectively. Memory is regarded as being fixed for any particular algorithm, and independent of both blocksize and bale size.

The Lifecycle of One Bale. Figure 2 shows the three phases in the lifecycle of each data bale, with time plotted on the horizontal axis and rate of expenditure (dollars per day, for example) plotted on the vertical axis. During *ingest*, arriving data is accumulated in uncompressed or simple-but-fast compressed form, and is not queryable. Once time t_1 is reached, this bale is closed to new data, and the permanent compression phase begins. At the same time, a new bale is opened to continue collecting data as it arrives; here we focus on just a single bale, noting that the same pattern of operations is repeated for each subsequent bale.

At time t_2 the bale has been compressed and the bale's block index has been constructed. At this time the uncompressed version of the bale is removed, and the long-term *service* phase commenced. During this period a decompression model (if the compression regime requires one) is held in main memory, and used to process access requests. Each access request also involves some CPU time and a disk operation. Finally, at time t_3, this bale reaches its retention date, and all of the resources associated with it are freed. To avoid clutter in the figure, disk access and disk transfer costs are not shown.

Rates of Expenditure. Starting at time t_0, the cost of storage increases broadly linearly according to the data arrival rate, λ, as shown in Figure 2. The SLA parameter L_W determines how much data can be included in this bale, because publication of the data and opening it up to queries requires that $t_2 \leq t_0 + L_W$. Moreover, since all data must be retained for at least L_D days, $t_3 \geq t_1 + L_D$. The difference $t_2 - t_1$ is the duration of the compression process, and can be shortened by allocating more concurrent processors – recall that each bale contains many blocks, and that the unit of compression and decompression is the block. That is, the width and height of the regions marked (B) and (C) in Figure 2 can vary according to the hardware used and the degree of parallelism employed, but the *areas* of those regions are constant, and hence their contributions to the total cost are determined solely by the size B of the bale.

Assembling the Parts. Given the variables listed in Table 1, it is clear that the total lifetime cost attributed to a bale is the sum of the dollar costs associated with each of the areas in Figure 2 – the total area under the curve – with expenditure rates (vertical axis) and time (horizontal axis) multiplied to yield costs measured in dollars. Component computations for these costs are shown in Figure 3, with all values expressed in units of "dollars per bale" over the lifetime of the bale, and with the balesize determined as $B = \lambda \cdot (t_1 - t_0)$. We noted earlier that Figure 2 did not show the IO costs associated with data accumulation and then querying. That simplification is rectified by including (A') and (D') respectively in Figure 3, to account for the reading and writing costs associated with secondary storage. The total retention cost for each bale of data is then given by

$$TRC(B) = \text{(A)} + \text{(A')} + \text{(B)} + \text{(C)} + \text{(D)} + \text{(D')} + \text{(E)} + \text{(F)},$$

measured, as already indicated, in units of "dollars per bale".

Implications. It was observed earlier that a less effective compression mechanism might have a lower TRC than a more effective one. Also worth noting are that employing parallelism during the conver-

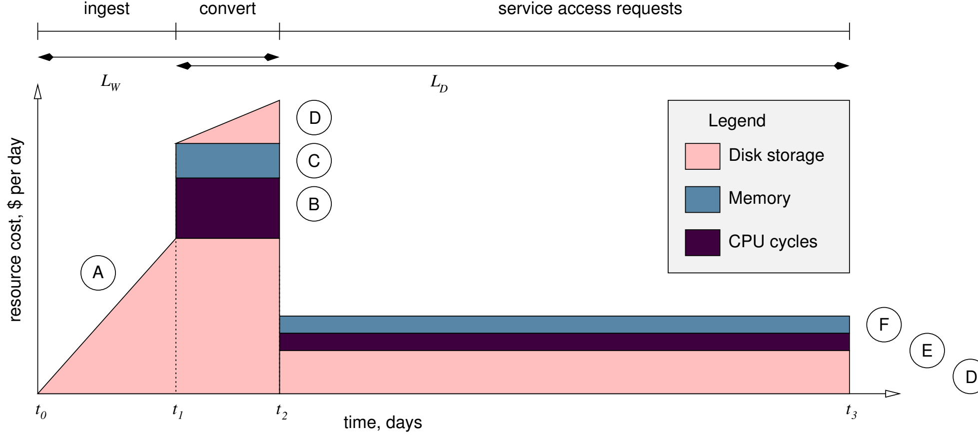

Figure 2: Lifecycle of a single bale: (A) cost of disk space for uncompressed data; (B) cost of CPU cycles required to compress the data; (C) cost of memory space needed during compression; (D) cost of disk space for compressed data; (E) cost of CPU cycles required to decompress data during access operations; (F) cost of memory space required by decoding model. The diagram is not to scale.

$$
\begin{aligned}
\text{(A)} &= (t_2 - t_1/2 - t_0/2) \cdot B \cdot C_{disk}/2^{10}/365 \\
\text{(A')} &= B \cdot 2 \cdot C_{io} \\
\text{(B)} &= \Big(B \cdot enc_{spd} + B/b \cdot enc_{ini} \cdot 2^{10}\Big) \cdot C_{cpu}/2^{20} \\
\text{(C)} &= (t_2 - t_1) \cdot enc_{mem} \cdot C_{mem}/2^{10}/365 \\
\text{(D)} &= (t_3 - t_2/2 - t_1/2) \cdot B \cdot ratio(b) \cdot C_{disk}/2^{10}/365 \\
\text{(D')} &= B \cdot ratio(b) \cdot C_{io} + q \cdot (t_3 - t_2) \cdot B \cdot b \cdot C_{io}/2^{20} \\
\text{(E)} &= q \cdot (t_3 - t_2) \cdot B \cdot (dec_{ini}/2^{10} + b/2^{20} \cdot dec_{spd}) \cdot C_{cpu}/2^{20} \\
\text{(F)} &= (t_3 - t_2) \cdot dec_{mem} \cdot C_{mem}/2^{10}/365
\end{aligned}
$$

Figure 3: Cost areas of components in Figure 2, together with the conversions required so that the units shown in Table 1 can be used to derive component costs in terms of dollars per bale.

Variable	Typical value
C_{cpu}	\$4.20 per PiCyc
C_{mem}	\$50.00 per GiB per year
C_{disk}	\$370.00 per TiB per year
C_{io}	\$0.00 per GiB

Table 2: Typical cloud computing costs, derived from Amazon Web Services pricing list.

Variable	Typical values			
	ZLib-9	xz-9	RLZ08	
enc_{mem}	1	70	45	MiB
enc_{ini}	0.08	45	0.24	MiCycles
enc_{spd}	140	850	760	Cycles per byte
dec_{mem}	0	0	8	MiB
dec_{ini}	0.08	0.08	0.24	MiCycles
dec_{spd}	14	37	14	Cycles per byte

Table 3: Algorithm performance variables, measured using the hardware described in Section 6. Both ZLib and xz were measured using the -9 option; and RLZ using an 8 MiB dictionary.

sion process may allow B to be larger, shifting t_1 closer to t_2 and hence reducing the number of bales requiring storage at any given moment in time; and that low query rates q and high latency limits L_R might mean that adaptive methods using large blocksizes can be compliant, removing some or all of (F), but increasing (E).

4. MODEL VERIFICATION

We now consider the various hardware-based and algorithm-based constants, in the middle two sections of Table 1.

Hardware Variables. Amazon Web Services (AWS) are a provider of cloud computing, offering a range of machine capacities at a range of dollar rates. We took their pricing tables[5] for “US West (Northern California)” and applied a rule-of-thumb that $2/3$ of the cost of the machine configurations listed were attributable to the CPU cycles that were included, and that the main memory accounted for $1/3$ of the net cost of each instance. For example, the “m4.large” instance is rated at 6.5 ECU (a comparative measure of CPU capacity, and broadly equivalent to a single 1 GHz processor), has 8 GiB of main memory, and costs \$0.14 per hour. We thus regard this cost as being a charge for main memory of $(1/3) \times \$0.14/8 = \$5.8 \times 10^{-3}$ per GiB per hour, or $C_{mem} \approx \$50$ per GiB per year. Carrying out the same arithmetic on other instance configurations suggests that this is a typical value. Similarly, we regard $(2/3) \times \$0.14/6.5 \approx \$1.4 \times 10^{-2}$ as the dollar cost of a single ECU-hour, which then corresponds to approximately \$4.20 per PebiCycle (that is, per 2^{40} clock cycles). This figure was also consistent over a range of instance options.

The EBS storage that can be associated with AWS instances is charged at a rate of \$0.03 per GiB-month, or \$370 per TiB-year, with no charge for data transfers to and from AWS computation devices. Table 2 lists the hardware costs used in our computations.

Algorithmic Variables. Table 3 lists the encoding and decoding performance of the three compression approaches explored in our experiments. All algorithm parameters were measured using in-

[5] https://aws.amazon.com/ec2/pricing/ and https://aws.amazon.com/ebs/pricing/, accessed 19 July 2016.

Method	(A)	(B)	(C)	(D)	(E)	(F)	$TRC(B)$
When $L_D = 365$, $q = 16$							
None	10	0	0	2316	0	0	2325
ZLib-64k	10	4	0	473	0	0	487
ZLib-1M	10	4	0	446	2	0	461
xz-64k	10	40	0	430	0	0	480
xz-1M	10	23	0	317	5	0	**356**
RLZ08-64k	10	20	0	327	0	39	395
RLZ08-1M	10	19	0	315	2	39	385
When $L_D = 365$, $q = 1024$							
None	10	0	0	2316	0	0	2325
ZLib-64k	10	4	0	473	9	0	495
ZLib-1M	10	4	0	446	132	0	591
xz-64k	10	40	0	430	22	0	503
xz-1M	10	23	0	317	347	0	697
RLZ08-64k	10	20	0	327	10	39	**405**
RLZ08-1M	10	19	0	315	133	39	516
When $L_D = 31$, $q = 16$							
None	10	0	0	200	0	0	209
ZLib-64k	10	4	0	41	0	0	54
ZLib-1M	10	4	0	38	0	0	**52**
xz-64k	10	40	0	37	0	0	87
xz-1M	10	23	0	27	0	0	60
RLZ08-64k	10	20	0	28	0	3	61
RLZ08-1M	10	19	0	27	0	3	60

Table 4: Total retention costs for Bale5, the sixth $B = 64$ GiB bale of HGOV2 (see Section 6 for details of the data, and for the compression ratios used), using three combinations of L_D and q, and two different blocksizes. The columns represent cost contributions in cents, summed over the lifetime of the bale, assuming the constants listed in Tables 2 and 3; the best TRC for each of the three parameter combinations is shown in bold. Note that (A') and (D') are both zero, and are not shown; and that the suffixes -64k and -1M on the methods refer to 64 kiB and 1,024 kiB blocksizes respectively.

memory computation averaged over 200 runs, thereby avoiding disk overheads. The fastest of the three is ZLib, which on our Intel Xeon E5640 2.67GHz machine encodes at around 20 MiB/sec and decodes at around 200 MiB/sec. The xz approach is the slowest, at around 3.1 MiB/sec encoding and 75 MiB/sec decoding. Even with the upfront cost of dictionary construction included, RLZ with an 8 MiB dictionary was measured at 3.6 MiB/sec encoding; it delivers the same high decoding rate as ZLib of 200 MiB/sec, confirming the measurements of Petri et al. [12]. The startup cost for each encoder was measured by encoding small 2–16 byte input files, and confirmed by subsequent experiments in which block sizes were varied. In particular, use of 64 kiB blocks substantially decreases the encoding speed of xz, down to 0.5 MiB/sec, whereas the encoding speed of ZLib and RLZ are relatively unaffected by block size. CPU cycle counts were also measured using the linux `perf stat` command. The cycles per byte values derived from the execution timings were in broad agreement with those reported by `perf`.

Memory usage was established by measuring peak `rss` memory (using the linux `getrusage()` function) while encoding and decoding 1 MiB blocks, excluding the input and output buffers.

SLA Constraints. The required query response time, L_R, is the factor most likely to influence the design of the data retention system. With plausible values for archived data being in the range of (say) 0.1 to 1 seconds, it is clear that the blocksize b must be small enough that the time taken to decode a data block is well under L_R, so that the time to access secondary storage can also be accommodated, possibly also including access to the bale's block index. Decoding rates (Table 3) for factor-based compression techniques exceed 75 MiB/second, and hence a blocksize of $b = 1$ MiB will require 0.01–0.02 seconds to decode. That is unlikely to be so large that the implementation will not be compliant.

Large blocks give the benefit of better compression ratios, even for RLZ-based schemes. The drawback of large blocks is that each query involves decoding a block of data to extract the required byte streams. In situations where the query rate q is high, methods with high decoding rates (small values of dec_{spd}) and/or smaller blocksizes b are likely to be more economical. On the other hand, when q is small (or zero), large block sizes and/or lower decoding rates can be tolerated in order to get improved compression ratios.

Example Cost Computation. Section 6 provides detailed results in regard to compression effectiveness. Table 4 employs some of those results in order to summarize the relative cost of different compression options. To construct the table, a bale of $B = 64$ GiB of web data partitioned into blocks of either $b = 64$ kiB or $b = 1$ MiB is assumed to be stored compressed using ZLib, using xz, and using the RLZ-ZZZ implementation of Petri et al. [10, 12] relative to an 8 MiB dictionary. The RLZ compression ratio is 5% absolute (around 25% relative) better than the ratio attained by ZLib, but some of the cost difference must be spent on main memory to store the dictionary, shown in column (F). When the query rate is low, in the top section of the table, the most economical approach is to use large blocks and the slow-but-good xz compressor. In the middle section of the table, query rates are assumed to be high, and the benefit of the RLZ approach can be seen – it provides fast decoding, and because it is semi-static, loses less compression effectiveness when applied to smaller blocks. In the bottom section of Table 4 the retention period is very short. Now it is the cheaper ZLib mechanism that has the lowest cost. Note that even in this short-term scenario, compression provides lower cost retention than uncompressed storage. Storing uncompressed data is unlikely to ever be cost-effective.

Key Factors. As a result of these explorations, we have identified three key factors that determine the relative cost of providing data retention and hence affect the choice of algorithm.

The first – already discussed – is the query rate. When high it creates downward pressure on the blocksize, and hence encourages use of a semi-static approach so as to maintain compression effectiveness. The second key factor is the retention period. The smaller the value of L_D, the more emphasis is placed on encoding costs and the less on compression ratio, and hence the extent to which slow-but-good algorithms can be cost-effective. The third factor is the relative cost of memory – when RAM is relatively cheap, the RLZ mechanism loses its disadvantage, and becomes the method of choice. The relationships between these three determining influences are illustrated in Figure 4.

5. REDUCING RETENTION COST

Based on the interactions between disk cost and memory cost illustrated in Table 4 and Figure 4, we now explore ways in which the TRC of the semi-static RLZ approach can be further reduced.

Multi-Dictionary Compression. To reduce the memory cost attributable to each bale, we propose the use of multiple shared dictionaries, with previous dictionaries retained beyond the end of their corresponding bales' lifetimes so that they can continue to be ex-

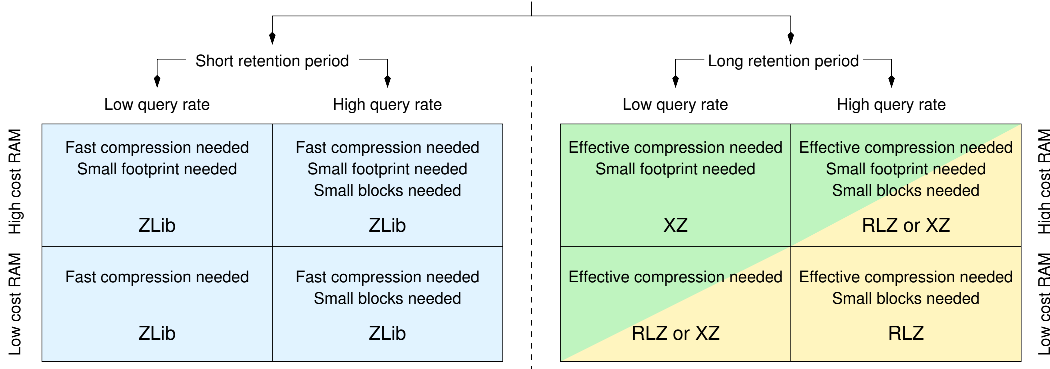

Figure 4: Key factors that determine the choice of compression mechanism: the retention period L_D; the query rate q; and the cost of RAM relative to the cost of disk storage.

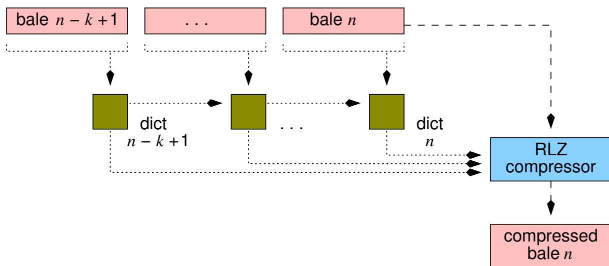

Figure 5: Using the dictionaries from k bales to compress bale n.

ploited. That is, instead of creating a dictionary of size dec_{mem} from the current nth bale, and attributing the cost of storing it solely to the nth bale, we compress the current bale relative to a composite dictionary of size $k \times dec_{mem}$ built from the last k bales, that is, bales $n-k+1$ to n, as shown in Figure 5. The cost of storing that dictionary can then be amortized across all of the k bales that share parts of it. The obvious way of forming a shared dictionary is to construct each section of it independently, and then simply concatenate the sections as required. But this has the drawback of allowing common strings to be selected into more than one of the component dictionaries, a wasteful duplication, since only one instance is required across the k dictionaries. Instead, we construct each dictionary adaptively, in the context of the previous $k-1$ dictionaries.

The dictionary-selection mechanism of Liao et al. [10] iteratively adds segments to the dictionary, choosing one high-scoring segment from each *epoch* of the input, with the epoch length chosen so as to result in a dictionary of the required size. The segment selection criterion is based on the estimated frequency in the bale – obtained from a sample – of the segment's m-grams. Once an m-gram is present in the dictionary as a consequence of it being included in a segment that has already been selected, subsequent iterations of segment selection can treat the frequency of that m-gram as zero. Extending this principle, when generating a dictionary for the nth bale, the m-grams in the concatenation of the $k-1$ previous bales' dictionaries are treated as having frequency zero in this nth bale. The dictionary generated as bale n's contribution will then be used to compress bale n and also the next $k-1$ bales. It will also influence the generation of the dictionaries for those upcoming bales.

Algorithm 1 illustrates this process, with *Bale* denoting the nth bale of the retention process, *Dicts* denoting a pool of previous dictionaries of which up to k (if that many are available) will be used,

Algorithm 1 Adaptive Dictionary Construction. A dictionary of size D and segment size s is to be constructed from *Bale*, by identifying D/s suitable segments, one from each epoch in *Bale*.

```
    function adaptive-dict(Bale, Dicts[max{0, n−k+1} ... n−1])
        Q ← extract-grams-union(Dicts)
        use reservoir sampler (parameter t = 512) to estimate
            the most frequent m-grams in Bale, together with their
5:          frequencies f(q)
        for all m-grams q ∈ Q do
            set frequency f(q) = 0       ▷ these ones already covered
        partition Bale into epochs of size Bs/D
        Dict ← ∅
10:     while |Dict| < D do
            pick at random an epoch E_i not yet considered
            for all segments S_j ∈ E_i do
                score S_j with the function g(·) in Liao et al. [10]
            add to Dict the segment in E_i with maximum g(·) score
15:         update the scoring function g(·)
        return Dict
```

Q storing the m-gram sets included in the previous dictionaries, and *Dict* denoting a growing pool of segments chosen from *Bale*. A reservoir sampler [9, 14] is used to estimate a set of high-frequency m-grams, and then each candidate segment is scored with regard to that set. The critical point is at step 7, where m-grams from previous dictionaries are prevented from influencing the scoring function, because those m-grams are already covered.

To support decoding using this multi-dictionary arrangement, the nth dictionary must be retained until bale $n+k-1$'s retention period is completed, that is, an extra $k-1$ bale cycles. Referring to Figure 2, zone (F) must extend beyond zone (E) by a further $k-1$ multiples of t_1-t_0. Hence zone (F)'s area increases by a factor of $(k-1)(t_1-t_0)/(t_3-t_2)$, where, as before, $t_0 < t_1 < t_2 \le t_0 + L_W$ and $t_1 + L_D \le t_3$. For example, assuming $t_3 - t_2 = L_D - (t_2 - t_1) = 366 - 1 = 365$ days, and $t_1 - t_0 = 1$ day, then this factor is $(k-1)/365$, less than 3% for values of k up to ten.

Accelerating Dictionary Construction. Although the dictionary-selection process shown in Algorithm 1 is effective [10], it is slow, because the scoring process embedded in the $g(\cdot)$ function sums m-gram frequencies for every candidate segment in *Bale*. In practice, we found that the earlier epochs contribute far more uncovered m-grams to *Dict* than do later epochs. To that end, it makes sense to

Bale	original order			homogenized		
	ZLib	xz	RLZ08	ZLib	xz	RLZ08
0 (64 GiB)	20.94	**16.27**	20.49	19.23	13.68	**13.57**
1 (64 GiB)	19.40	13.30	**12.79**	19.24	13.70	**13.57**
2 (64 GiB)	16.69	10.07	**6.88**	19.24	13.70	**13.58**
3 (64 GiB)	15.77	8.35	**4.33**	19.24	13.69	**13.57**
4 (64 GiB)	15.26	7.91	**4.13**	19.25	13.70	**13.59**
5 (64 GiB)	20.36	**13.08**	14.95	19.26	13.71	**13.60**
6 (42 GiB)	26.34	**21.22**	30.45	19.24	13.70	**13.57**
Average	19.25	**12.89**	13.43	19.24	13.70	**13.58**

Table 5: Compression ratios (percent relative to original size) for non-randomized and randomized versions of the GOV2 collection, with bale size $B = 64$ GiB, blocksize $b = 1$ MiB and RLZ decoding dictionary of $dec_{mem} = 8$ MiB. The best ratios are in bold.

front-load the segment-addition procedure: the algorithm becomes faster, while the dictionary can still provide good coverage of the bale. To this end, we keep the segment size s unchanged, and shrink the epoch size (step 8) to a fraction of Bs/D. Then, once the dictionary reaches size D, the remaining unprocessed epochs are ignored.

In preliminary experiments we found that if D/B is around 0.01% (a 640 kiB dictionary for a 64 GiB bale), then the epoch size could be shrunk by a factor of 16, while if D/B is around 0.1%, then the epoch size could be shrunk by a factor of 8, with a drop in compression effectiveness of only around 0.1% (in absolute terms) on the HGOV2 and GOV2 collections (see Section 6); both with a ten-fold increase in encoding throughput. This approach was used in all of the experimentation, including in Table 3.

6. EXPERIMENTS

Hardware and Methodology. All of the RLZ methods are implemented using `c++11` and compiled with `gcc` 5.2.1 running on a linux server equipped with 148 GiB RAM and an Intel E5640 processor. Much of the functionality we employ is built on top of the SDSL succinct data structures library described by Gog et al. [4]. In all experiments with RLZ we use a segment size of $s = 1{,}023$ bytes, the dictionary selection process of Liao et al. [10] including random-epoch ordering, the -ZZZ modifications described by Petri et al. [12], and the speedup procedure described in Section 5.

Dataset. We use the widely available 426 GiB GOV2 web collection as a basis for our experiments. It contains a 2004 crawl of the `.gov` websites, covering around 25,000,000 pages, primarily html and pdf documents, each truncated at 256 kiB.[6] To avoid the localized document clustering inherent in a one-off web-crawl, and to better reflect the ongoing situation we seek to model and report on, we generated a random permutation of the documents, and took that ordering to be the data arrival sequence for the purpose of generating bales to form a new collection, HGOV2. Table 5 shows the result of applying ZLib, xz, and RLZ to bales of $B = 64$ GiB using the original document ordering and the homogenized ordering. The homogenization process gives each bale similar properties, as would be expected in a data retention context. Note the relatively similar compression ratios obtained by RLZ08 and xz, but that RLZ executes more quickly for both encoding and decoding (Table 3), even though it makes two passes.

[6] See http://ir.dcs.gla.ac.uk/test_collections/gov2-summary.htm.

Method	Blocksize b		
	64 kiB	256 kiB	1024 kiB
ZLib	20.44	19.50	19.26
xz	18.57	16.15	13.71
RLZ01	17.40	16.88	16.74
RLZ02	16.31	15.84	15.71
RLZ04	15.18	14.74	14.62
RLZ08	14.10	13.71	13.60
RLZ16	13.18	12.83	12.72
RLZ32	12.48	12.15	12.04
RLZ64	**11.82**	**11.50**	**11.39**

Table 6: Measured compression effectiveness on Bale5 of the HGOV2 collection, for three different blocksizes and a range of compression mechanisms. Values are overall percentages relative to the original size; the best compression is achieved by RLZ with a large dictionary. The per-block compression ratios plotted in Figure 1 were drawn from this data.

Method	Blocksize b		
	64 kiB	256 kiB	1024 kiB
ZLib	487	465	461
xz	480	411	**356**
RLZ01	437	425	424
RLZ02	417	406	405
RLZ04	400	391	389
RLZ08	**395**	**386**	385
RLZ16	413	405	404
RLZ32	475	467	466
RLZ64	616	608	607

Table 7: Total retention costs in cents over the lifetime of Bale5 of the HGOV2 collection, for three different blocksizes, assuming a retention duration of $L_D = 365$ days, a query rate of $q = 16$ queries per GiB per day, the compression ratios shown in Table 6, and other parameters as shown in Tables 2 and 3. The best value in each column is shown in bold.

Baseline Compression. Table 6 lists compression ratios for Bale5 of HGOV2 for three different blocksizes and a range of compression methods. The best compression is obtained with RLZ, a large blocksize, and a large dictionary. Table 7 then translates those compression effectiveness measurements into TRC values, assuming (as in the top section of Table 4) that $L_D = 365$ days and a query rate of $q = 16$ queries per GiB per day. In this experiment xz provides the smallest TRC, beating RLZ08 by around 10%. But once the query rate reaches $q = 128$, the RLZ08 mechanism becomes the cheapest, because it provides better compression when blocks smaller than 1 MiB are employed. The middle section of Table 4 shows the clear advantage that RLZ enjoys for high query rates.

Multi-Dictionary Compression. Table 8 shows the gain in compression possible via the multi-dictionary approach. For example, if 2 MiB of dictionary per bale is combined, then a "bales 2+3+4+5" dictionary gives the same compression effectiveness (14.11%) as a single "Bale5 only" dictionary of 8 MiB (Table 6, 14.10%). With this modification, and the same settings as assumed in the top section of Table 4, the least cost mechanism is the RLZ04 mechanism using a 6-bale dictionary of 24 MiB in total, constructed by amalgamating six 4 MiB dictionaries. Each dictionary is now required

Method	Bales used to construct dictionary		
	4+5	2+3+4+5	0+1+2+3+4+5
RLZ01	16.26	15.16	14.53
RLZ02	15.18	14.11	13.56
RLZ04	14.11	13.18	12.82
RLZ08	13.19	12.48	12.07
RLZ16	12.48	11.81	11.43
RLZ32	11.82	11.19	10.81
RLZ64	**11.17**	**10.52**	**10.15**

Table 8: Compression effectiveness when multi-bale dictionaries are used, assuming $L_D = 365$ days. All results are for Bale5 of HGOV2 using $b = 64$ kiB blocks. These values can be directly compared with those shown in Table 6.

to be present for 370 days rather than 365, a 1.3% overhead on the memory required if it is assumed that $\lambda = 64$ GiB/day and each bale is one day's worth of retained data. In this combination, the cost is 346 cents per bale, under-cutting the 356 cents per bale shown for xz in Tables 4 and 7. That is, the multi-dictionary approach shifts the trade-off point between xz and RLZ in favor of RLZ. With higher querying rates, the advantage is similarly greater. The trade-off balance between RLZ and ZLib is also affected, but to a lesser extent. With $L_D = 31$ (the lower section of Table 4), the cost of RLZ04 using a combined 24 MiB dictionary is reduced slightly to 56 cents per bale, and ZLib is still fractionally cheaper. Part of the reason why multi-dictionaries are less effective for such short retention periods is that now the overhead on memory cost is $(36-31)/31 = 16.1\%$.

Storing the Dictionary on Secondary Storage. The evaluations shown above assume that the RLZ dictionary is retained in memory, and that the bale's block index is stored on disk or SSD, and can be probed to identify the address of the compressed block needed by each query operation. (Note that this does not alter the comparisons, as the same block table is required for all methods.) If the SLA parameter L_R permits, the RLZ dictionary might also be stored on disk or SSD. If so, a query operation would then consist of five steps: use the block index to identify the disk address of the compressed block; fetch that block; fetch the dictionary associated with the bale; perform the decoding; and extract and return the desired bytes. The first three of these would each involve a disk seek operation; and the third might also involve transfer of a substantial volume of data. Given the gains already achieved by the multi-dictionary approach, the additional TRC reductions would be relatively small; nevertheless, we note this option as a further possibility if L_R permits three disk accesses, plus transfer of a dozen or more MiB of data.

URL Sorting. Sorting web collections by URL is known to enhance compressibility, and the GOV2 and HGOV2 collections are no exception. In the case of the HGOV2 collection, if we assume that the unit of access is a document, then an additional document permutation index containing approximately three million "docnum, block-num"' pairs must be maintained for each bale, or around 36 MiB. If this index can also be stored on disk or SSD, then the additional CPU time spent during the publication phase to sort each of the bales leads to substantial reductions in TRC for the adaptive compression tools. For example, a URL-sorted version of Bale5 from HGOV2 represented with 1 MiB blocks obtains a compression ratio of 11.50% with ZLib, 6.20% with xz; and 10.81% with RLZ08. In the case of xz, the TRC per bale would approximately halve. Note that this option is specific to web-crawl data.

RLZ Variants. As a final comment, we note that xz and Bzip2 can be used as back-ends with the RLZ approach, to obtain further slight compression gains and hence potentially offer other TRC trade-offs. For example, a RLZ-XXX version using a 8 MiB dictionary, xz as a back-end coder, and 1 MiB blocks, reduces Bale5 of HGOV2 to 11.32%, and the URL-sorted version of that bale to 8.44%.

7. RELATED WORK

Computing in the Cloud. In 2003 Jim Gray summarized the changing economics of computing with the growth of cloud computing (then called Internet-scale distributed computing) [6]. Many subsequent papers have analyzed the cost of deploying computation – databases, scientific computation, a backed-up file system, oblivious storage – in the cloud. Chen and Sion [3] examine the costs of transferring data and computation from one venue, such as a home, or small-to-medium enterprise, to the cloud. Their contention is that disk is extremely cheap: given the expense of transfer, only CPU-intensive computations should be moved to the cloud. However, most references, as well as the AWS pricing schemes, suggest that disk is of the order of a few hundred dollars per terabyte-year, relegating transfer to around a quarter of the cost of storing a byte for a year. Although a 1 TiB disk can be purchased for around $100, the cost of maintaining the disk (including its power), and of storing redundant copies means that the $370/ TiB-year is reasonable [1].

The same Berkeley Technical Report focuses on transfer rate as a limiting factor in deploying scientific computation in the cloud [1]. In particular, Armbrust et al. consider an example of a biology lab transferring half a terabyte of data to a cloud server, taking 55 hours. It turns out that sending disks via courier to Amazon and accounting for the labor costs might have the least relative latency, and is possibly the cheapest solution. Monitoring AWS prices between 2006 and 2008, Armbrust et al. also observe that disk storage prices were stable, but network traffic dropped in price. Meanwhile, the trade-off between computing power per dollar and memory per dollar depends on the machine.

Privacy-preserving (oblivious access) data structures are claimed to be an important part of secure cloud computing. On top of asymptotic analysis, Goodrich et al. [5] analyze the monetary cost of their data structures. However, compared with our work, they focus on transfer rates and costs of transferring data to Amazon S3 storage. Bindschaedler et al. [2] follow this up with an extensive analysis of oblivious access on the cloud, running experiments on large file-system benchmark, again focusing on put and get requests.

Shi et al. [13] propose a full file system, Saga, backed up by Amazon S3 and EC2. To minimize transfer cost, they employ a cache, especially to cope with the expense of Put requests. Their system has block sizes between 512 kiB and 4 MiB, and employs block-level deduplication, a coarse compression scheme. Shi et al. find that adding Bzip2 to the process slows reading and writing considerably, especially for larger blocks. Although they observe a reduction in cloud storage consumed, they do not trade these quantities off economically, as we do.

Cost of Compression. Zohar and Cassuto [16] minimize the cost of compression via a general optimization framework for selecting the best of a number of compression schemes. Assuming a system with very limited computing resource, they make decisions about whether to compress a file on an individual basis, or on a small group of files. With an elastic computing system available, however, our focus is on the choice of a single algorithm, assuming the economy of scale of a single compression scheme for a large volume of data.

8. CONCLUSIONS

We have provided a detailed cost model for compression-based data retention schemes, and shown that a range of factors influence the total cost of archiving data, including the retention period, the rate of access requests, and the relative cost of main memory. In particular, different compression schemes should be used in different operating environments. Guided by the cost model, we have described an improved RLZ mechanism that reduces the monetary costs associated with data retention in a wide range of situations, and increases the size of the parameter-space region in which RLZ is the least-cost option.

A further enhancement that we plan to pursue is to combine semi-static and adaptive compression, allowing factors to be drawn both from the k previous dictionaries, and also from the part of the block that has already been represented. The idea is to seed the block's model with the static dictionary, so that good effectiveness is possible right from the beginning; plus incorporate localized learning that will allow a focus on the content of this particular block.

Software. The implementation used to generate the experimental results is available at https://github.com/unimelbIR/rlz-store.

Acknowledgments. This work was supported under Australian Research Council's Discovery Projects funding scheme (project number DP140103256).

References

[1] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. H. Katz, A. Konwinski, G. Lee, D. A. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. Above the clouds: A Berkeley view of cloud computing. Technical Report UCB/EECS-2009-28, University of California, Berkeley, Feb. 2009.

[2] V. Bindschaedler, M. Naveed, X. Pan, X. Wang, and Y. Huang. Practicing oblivious access on cloud storage: The gap, the fallacy, and the new way forward. In *Proc. ACM Conf. on Computer and Communications Security (SIGSAC)*, pages 837–849, 2015.

[3] Y. Chen and R. Sion. To cloud or not to cloud? Musings on costs and viability. In *Proc. ACM Symp. on Cloud Computing (SoCC)*, pages 29:1–29:7, 2011.

[4] S. Gog, T. Beller, A. Moffat, and M. Petri. From theory to practice: Plug and play with succinct data structures. In *Proc. Symp. on Experimental and Efficient Algorithms (SEA)*, pages 326–337, 2014.

[5] M. T. Goodrich, M. Mitzenmacher, O. Ohrimenko, and R. Tamassia. Practical oblivious storage. In *Proc. ACM Conf. on Data and Application Security and Privacy (CODASPY)*, pages 13–24, 2012.

[6] J. Gray. Distributed computing economics. Technical Report MSR-TR-2003-24, Microsoft Research, Mar. 2003.

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

[8] P. Kulkarni, F. Douglis, J. D. LaVoie, and J. M. Tracey. Redundancy elimination within large collections of files. In *Proc. USENIX Ann. Tech. Conf., Gen. Trk.*, pages 59–72, 2004.

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

[10] K. Liao, M. Petri, A. Moffat, and A. Wirth. Effective construction of Relative Lempel-Ziv dictionaries. In *Proc. Conf. on the World Wide Web (WWW)*, pages 807–816, 2016.

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

[12] M. Petri, A. Moffat, P. C. Nagesh, and A. Wirth. Access time tradeoffs in archive compression. In *Proc. Asia Information Retrieval Societies Conf. (AIRS)*, pages 15–28, 2015.

[13] W. Shi, D. Ju, and D. Wang. Saga: A cost efficient file system based on cloud storage service. In *Proc. Wrkshp. on Economics of Grids, Clouds, Systems, and Services (GECON)*, pages 173–184, 2011.

[14] J. S. Vitter. Random sampling with a reservoir. *ACM Trans. on Mathematical Software*, 11(1):37–57, 1985.

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

[16] E. Zohar and Y. Cassuto. Data compression cost optimization. In *Proc. IEEE Data Compression Conf. (DCC)*, pages 393–402, 2015.