Differential Erasure Codes for Efficient Archival of Versioned Data in Cloud Storage Systems

J. Harshan(✉), Anwitaman Datta, and Frédérique Oggier

Nanyang Technological University, Singapore, Singapore
jharshan@ntu.edu.sg

Abstract. In this paper, we study the problem of storing an archive of versioned data in a reliable and efficient manner. The proposed technique is relevant in cloud settings, where, because of the huge volume of data to be stored, distributed (scale-out) storage systems deploying erasure codes for fault tolerance is typical. However existing erasure coding techniques do not leverage redundancy of information across multiple versions of a file. We propose a new technique called differential erasure coding (DEC) where the differences (deltas) between subsequent versions are stored rather than the whole objects, akin to a typical delta encoding technique. However, unlike delta encoding techniques, DEC opportunistically exploits the sparsity (i.e., when the differences between two successive versions have few non-zero entries) in the updates to store the deltas using sparse sampling techniques applied with erasure coding. We first show that DEC provides significant savings in the storage size for versioned data whenever the update patterns are characterized by in-place alterations. Subsequently, we propose a practical DEC framework so as to reap storage size benefits against not just in-place alterations but also real-world update patterns such as insertions and deletions that alter the overall data sizes. We conduct experiments with several synthetic and practical workloads to demonstrate that the practical variant of DEC provides significant reductions in storage-overhead.

Keywords: Cloud storage · Backup and recovery · Fault tolerance · Erasure coded storage

1 Introduction

Over the last decade, cloud storage services have revolutionized the way we store and manage our digital data. Due to massive advances in the internet-, wireless-, and storage-technologies, plenty of file hosting services are nowadays offering storage and/or computing facilities to store huge amounts of data that are accessible from various locations on different devices and platforms. From an engineering view point, developing such ubiquitous cloud storage services necessitates in-depth understanding of, (i) *reliability:* how to store data across a network by guaranteeing a certain fault tolerance against failure of storage devices? (ii) *security:* how to protect data from security threats, both by a passive

A. Hameurlain et al. (Eds.): TLDKS XXX, LNCS 10130, pp. 23–65, 2016.
DOI: 10.1007/978-3-662-54054-1_2

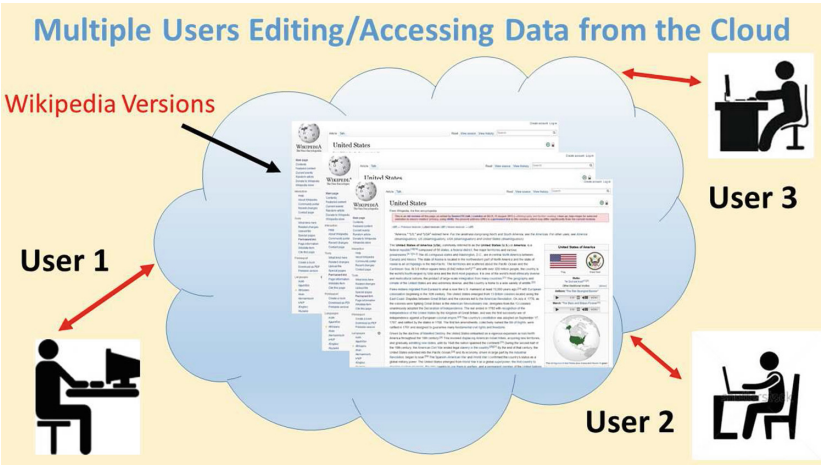

Fig. 1. Schematic depicting typical cloud storage application where multiple users access data from the cloud facility. In this illustrative figure, users are accessing and editing Wikipedia pages thereby creating a repository of multiple versions. Our work addresses a new framework of distributed storage systems for versioned data, aiming to lay foundations to new system architectures for cloud storage utilities supporting multiple versions.

adversary which is interested in reading the stored data, and also by an active one which is keen on manipulating the existing data? and (iii) *availability:* how to spread data across the network so as to speed up synchronization between client devices and the cloud. Each of the above aspects is a specialized area of study in this field, and all have been active areas of research.

In this work, we address the reliability aspect in cloud storage wherein we are interested in developing efficient ways of storing, accessing and modifying data in the cloud, while at the same time guaranteeing a certain level of fault tolerance against device (node) failures. In cloud storage networks, redundancy of the stored data is critical to ensure fault tolerance against node failures. While data replication remains a practical way of realizing this redundancy, the past years have witnessed the adoption of erasure codes for data archival, e.g. in Microsoft Azure [1], Hadoop FS [2], or Google File System [3], which offer a better trade-off between storage-overhead and fault tolerance. Thus, design of erasure coding techniques amenable to reliable and efficient storage has accordingly garnered a huge attention [5,9]. Once the reliability aspect of erasure codes for standalone objects is better understood, it is natural to question the reliability of versioned data. The need to store multiple versions of data arises in many scenarios. For instance, when editing and updating files, users may want to explicitly create a version repository using a framework like SVN [6]. Cloud based document editing or storage services also often provide the users access to older versions of the documents, e.g., Google Docs, Microsoft's Office 365, and Apple's iWork. See Fig. 1 for an illustrative example where multiple users access/modify data

on Wikipedia thereby creating a versioned repository. Another scenario is that of system level back-up, where directories, whole file systems or databases are archived - and versions refer to the different system snapshots. Example systems include Dropbox which provides backup storage over which several users can collaborate creating multiple versions of data. In either of the two file centric settings, irrespective of whether a working copy used during editing is stored locally or on the cloud, or in a system level back up, say using copy-on-write [7], the back-end storage system needs to preserve the different versions reliably, and can leverage on erasure coding for reducing the storage-overheads.

1.1 Significance and Applications

It is well known that versioning systems get rid of duplicated contents across subsequent versions of a file. The main objective of versioning is to store only the changes from the preceding versions so as to reduce the overall storage size, and yet be able to accurately reconstruct any version requested by a user. In general, versioning concept falls within a broad topic of deduplication, that works on a plethora of files over space and time, and not only on temporal changes of a file. Meanwhile, in distributed storage systems erasure coding has gained enormous attention as it provides reduced storage-overhead when compared with the replication scheme. Although erasure coding schemes and architectures have been applied on standalone data objects in the past, the literature on erasure coding to versioned data is scarce. One possible reason for scarcity might be the possibility of a straightforward option to apply erasure coding on the changes (deltas), i.e., to treat versioning and erasure coding as independent entities. In this work, we explore a new direction to develop a close-coupled compression and erasure coding technique that can reduce the complexity of the versioning system and still yield high fault tolerance and significant storage gains. This work develops on our preliminary work in [18], where an erasure coding technique was proposed to reduce the I/O gains when retrieving multiple versions of data.

Applications for retrieving versioned archive include software development environments wherein multiple versions of modules are developed by different members of the project, and are often checked into the system at different time instants, e.g. management of software files over CVS. In such applications, the system administrator or the project/team lead would need to retrieve multiple versions at once in order to perform consistency checks and/or to possibly merge the contents based on the nature of changes. In back-up applications like "time machine"(where there is no revision history, etc. in contrast to a SVN like application), even if the user may eventually check out a few random versions to locate the version they want, it is often desirable to prefetch several subsequent versions so that the user can browse through them and also navigate consecutive versions to identify the one that is finally needed. There, our strategy will have a superior performance.

1.2 Related Works

Erasure codes have been extensively deployed in practical distributed storage systems for efficient and reliable storage of data [8]. Their choice over the standard replication technique comes as a natural course of action since erasure codes were proven to reduce the storage-overhead while maintaining a given fault tolerance level. However, in the recent past, with the objective of maintaining storage systems intact despite high failure rate of devices, [4], a plethora of new erasure code constructions have surfaced to not only reduce the storage-overhead but also facilitate low-complexity repair process for recovering lost data [1]. Since then most works have focused on distributed storage architectures for storing stand-alone objects, and not many have addressed the aspect of efficiently storing multiple versions of data. The topic of erasure coding for versioned data is loosely related to the issues of efficient updates [12–15], and of deduplication [16], which is the process of eliminating duplicate data blocks in order to eliminate unnecessary redundancy. Existing works on update of erasure coded data focus on the computational and communication efficiency in carrying out the updates, with the goal to store only the latest version of the data, and thus do not delve into efficient storage or manipulation of the previous versions. Recently, Wang and Cadambe [10] have addressed multi-version coding for distributed data, where the underlying problem is to encode different versions so that certain subsets of storage nodes can be accessed to retrieve the most common version among them. Their strategy has been shown applicable when the updates for the latest version do not reach all the nodes, possibly due to network problems. More recently, in [11], the authors have considered the problem of synchronizing data in storage networks under an edit model that includes deletions and insertions. They propose several variants of erasure codes that allow updates on the parity check values with low-bit rates and small storage-overhead. Apart from [10,11] that nearly touch upon the subject of storing versioned data, not many contributions exist in the literature that explicitly address erasure coding schemes for versioned data.

Capitalizing on the advances in erasure coding techniques for distributed storage, a straightforward option is to apply erasure coding on deltas of a versioning scheme. One such well-known scheme is *Rsync* [19], which is widely used for file transfer and synchronization across networks. The key idea behind Rsync is the rolling checksum computation, using which only the modified/new blocks between successive versions are transferred, thereby reducing the communication bandwidth. When such algorithms are applied to store versioned data, significant reduction in storage size is expected. Thus, Rsync scheme indirectly falls in the related works section of this topic. In [18] (extended abstract available in [17]), we have proposed erasure codes for storing multi-versioned data to benefit purely in terms of I/O, and for objects of fixed size. This paper develops on [18] to not only provide I/O benefits, but also to yield total storage savings when storing different versions of data. Furthermore, various system-level implementation issues of this work have been discussed in [24]. So this contribution distinguishes from [18,24] by exploring new erasure coding techniques that suit

the underlying versioning model. In the next section, we summarize the key idea of this paper.

1.3 The Key Idea

Sparse Signal Recovery (SSR) [20] has garnered widespread applications as a powerful signal processing technique that can extract and store sparse signals with significantly fewer measurements than its conventional counterparts. In this paper, we explore how SSR ideas can be adapted by the storage community to facilitate reduced storage size for big data in cloud storage networks. In this context, the word *signal* refers to a vector of real numbers (with respect to some basis), whereas a *sparse signal* refers to one with fewer non-zero components compared to the length of the vector. Some applications of SSR include audio and video processing, medical imaging, and communication systems, where the signal acquisition process projects the desired sparse signal into a lower-dimensional space, and then appropriately recovers the higher-dimensional sparse signal from the lower-dimensional signal. Although SSR techniques are widely applied to signals over real numbers, this topic is also extendable to finite fields [21]. An illustrative example is discussed in Fig. 2. Now, the reader may ask, how does one obtain sparse vectors in storage systems? The answer lies in the fact that when different versions of data object are stored, there is a possibility of a user introducing few changes between subsequent versions, which in turn may result in sparse difference vectors.

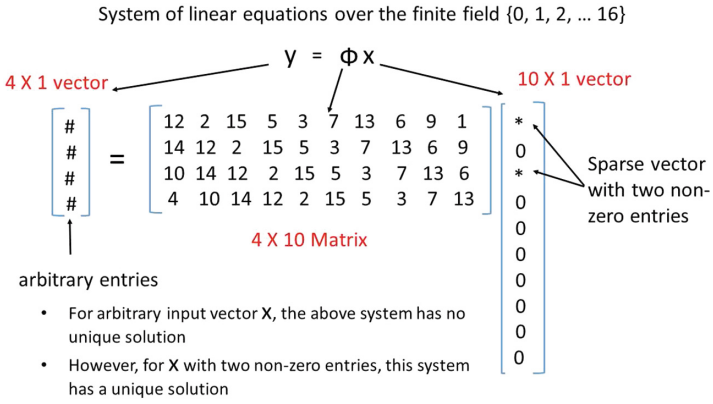

Fig. 2. An example to illustrate the possibility of recovering a sparse vector from a lower-dimensional vector. In this example, input vector which is 2-sparse (only two non-zero components irrespective of the positions) can be accurately recovered with 4 observations over the finite field $\mathbb{F}_{17} = \{0, 1, 2, \cdots, 16\}$, where arithmetic operations are over modulo 17.

1.4 Contributions

The contributions of this work are summarized below:

- We propose a new differential erasure coding (DEC) framework that falls under the umbrella of delta encoding techniques, where the differences (deltas) between subsequent versions are stored rather than the whole objects. The proposed technique exploits the sparsity in the differences among versions by applying techniques from sparse sampling [20], in order to reduce the storage-overhead (see Sects. 2 and 3). We have already proposed the idea of combining sparse sampling with erasure coding in [18], where we studied the benefits purely in terms of I/O, and for objects of fixed size. While retaining the combination of sparse sampling and erasure coding, this work introduces a different erasure coding strategy which provides storage-overhead benefits.
- We first present a simplistic layout of DEC that relies on fixed object lengths across successive versions of the data, so as to evaluate the right choice of erasure codes to store versioned data. We show that when all the versions are fetched in ensemble, there is also an equivalent gain in I/O operations. This comes at an increased I/O overhead when accessing individual versions. We accordingly propose some heuristics to optimize the basic DEC, and demonstrate that they ameliorate the drawbacks adequately without compromising the gains at all. Further, we show that the combination of sparse sampling and erasure coding yields other practical benefits such as the possibility of employing fewer erasure codes against different sparsity levels of the update patterns. (see Sect. 4).
- In the later part of this paper, we extend the preliminary ideas of DEC to develop a framework for practical DEC that is robust to real-world update patterns across versions such as insertions and deletions which may alter the overall size of the data object. Along that direction, we acknowledge that insertions and deletions may ripple changes across the object at the coding granularity, and may also increase the object size. Such rippling effect could in particular render DEC useless, and obliterate the consequent benefits. We apply the zero-padding idea introduced in [24] to ameliorate the aforementioned problems of insertions and deletions (see Sect. 5).
- For storing versioned data, the total storage size for *deltas* inclusive of zero pads (prior to erasure coding) is used as the metric to evaluate the quantum and placement of zero pads against a wide range of workloads that include insertions and deletions, both bursty and distributed in nature (see Sect. 6). We compare the storage savings offered by the practical DEC technique with the standard baselines against both synthetic and practical datasets. The baselines include (i) a non-differential scheme, where different versions of a data object are encoded in full without exploiting the sparsity across them, (ii) selective encoding scheme, wherein only modified blocks between the versions are stored, (iii) Rsync, a delta encoding technique for file transfer and synchronization across networks, and (iv) gz compression algorithm applied on individual versions to reduce the storage size of each version. Among the

four baselines, we show that DEC outperforms (i), (ii) and (iv), in terms of storage size, while trades-off storage size to computational complexity when compared with (iii). We use Wikipedia datasets to showcase the impact of DEC scheme on practical datasets.

1.5 System Model for Version Management

Any digital content to be stored, be it a file, directory, database, or a whole file system, is divided into data chunks, shown as phase ① in Fig. 3. The proposed coding techniques are agnostic of the nuances of the upper levels, and all subsequent discussions will be at the granularity of these chunks, which we will refer to as data objects or just objects.

Formally, we denote by $\mathbf{x} \in \mathbb{F}_q^k$ a data object to be stored over a network, that is, the object is seen as a vector of k blocks (phase ②) taking value in the alphabet $\mathbb{F}_q$, with $\mathbb{F}_q$ the finite field with q elements, q a power of 2 typically. Encoding for archival of an object $\mathbf{x}$ across n nodes is done (phase ③) using an (n, k) linear code, that is $\mathbf{x}$ is mapped to the codeword

$$\mathbf{c} = \mathbf{G}\mathbf{x} \in \mathbb{F}_q^n, \ n > k, \tag{1}$$

for $\mathbf{G}$ an $n \times k$ generator matrix with coefficients in $\mathbb{F}_q$. We use the term *systematic* to refer to a codeword $\mathbf{c}$ whose k first components are $\mathbf{x}$, that is $c_i = x_i$, $i = 1, \ldots, k$. This described what is a standard encoding procedure used in erasure coding based storage systems. We suppose next that the content mutates, and we wish to store all the versions.

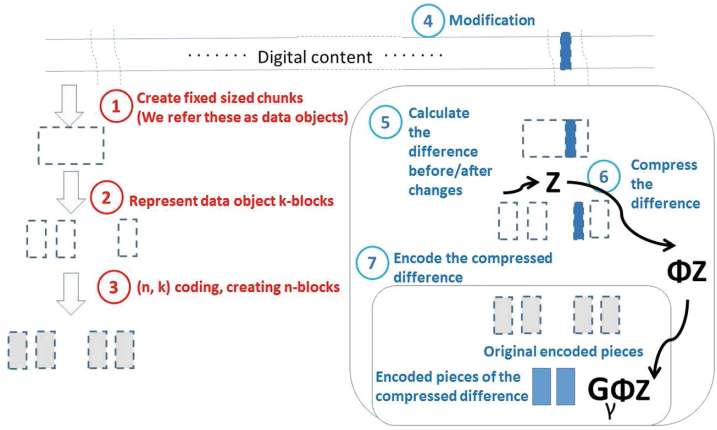

Fig. 3. An overview of the coding strategy using compressed differences

Let $\mathbf{x}_1 \in \mathbb{F}_q^k$ denote the first version of a data object to be stored. When it is modified (phase ④), a new version $\mathbf{x}_2 \in \mathbb{F}_q^k$ of this object is created. More

generally, a new version $\mathbf{x}_{j+1}$ is obtained from $\mathbf{x}_j$ to produce over time a sequence $\{\mathbf{x}_j \in \mathbb{F}_q^k,\ j = 1, 2, \ldots, L < \infty\}$ of L different versions of a data object, to be stored in the network. We are not concerned with the application level semantic of the modifications, but with the bit level changes in the object. Thus the changes between two successive versions are captured by the relation

$$\mathbf{x}_{j+1} = \mathbf{x}_j + \mathbf{z}_{j+1}, \tag{2}$$

where $\mathbf{z}_{j+1} \in \mathbb{F}_q^k$ denotes the modifications (in phase ⑤) of the jth update. We first assume fixed object lengths across successive versions of data so as to build an uncomplicated framework for the differential strategy. Such a framework shields us from unnecessarily delving into system specificities, instead, serves as a foundation to evaluate various erasure coding techniques to store multiple versions of data. We show that the design, analysis and assessment of the coding techniques are oblivious to the nuances of how the data object is broken down into several chunks prior to the encoding purposes, thereby facilitating us to segregate *chunk synthesis* and *erasure coding* blocks as two independent entities. In the later part of this work (see Sect. 5), we discuss how to relax the fixed object length assumption and yet develop a practical DEC scheme, that is robust to variable object lengths across successive versions.

The key idea is that when the changes from $\mathbf{x}_j$ to $\mathbf{x}_{j+1}$ are small (decided by the sparsity of $\mathbf{z}_{j+1}$), it is possible to apply sparse sampling [20], which permits to represent a k-length γ-sparse vector $\mathbf{z}$ (see Definition 1) with less than k components (phase ⑥) through a linear transformation on $\mathbf{z}$, which does not depend on the position of the non-zero entries, in order to gain in storage efficiency.

Definition 1. *For some integer $1 \leq \gamma < k$, a vector $\mathbf{z} \in \mathbb{F}_q^k$ is said to be γ-sparse if it contains at most γ non-zero entries.*

Let $\mathbf{z} \in \mathbb{F}_q^k$ be γ-sparse such that $\gamma < \frac{k}{2}$, and $\Phi \in \mathbb{F}_q^{2\gamma \times k}$ denote the *measurement matrix* used for sparse sampling. The compressed representation $\mathbf{z}' \in \mathbb{F}_q^{2\gamma}$ of $\mathbf{z}$ is obtained as

$$\mathbf{z}' = \Phi \mathbf{z}. \tag{3}$$

The following proposition[1] gives a sufficient condition on Φ to uniquely recover $\mathbf{z}$ from $\mathbf{z}'$ using a syndrome decoder [21, Sect. II.B].

Proposition 1. *If any 2γ columns of Φ are linearly independent, the γ-sparse vector $\mathbf{z}$ can be recovered from $\mathbf{z}'$.*

Once sparse modifications are compressed, which reduces the I/O reads, they are encoded into codewords of length $< n$ (phase ⑦) decreasing in turn the storage-overhead.

[1] The proof for the proposition follows from the property that any 2γ columns of a parity check matrix of a linear code with minimum distance $2\gamma + 1$ are linearly independent.

2 Differential Erasure Encoding for Version-Control

Let $\{\mathbf{x}_j \in \mathbb{F}_q^k,\ 1 \leq j \leq L\}$ be the sequence of versions of a data object to be stored. The changes from $\mathbf{x}_j$ to $\mathbf{x}_{j+1}$ are reflected in the vector $\mathbf{z}_{j+1} = \mathbf{x}_{j+1} - \mathbf{x}_j$ in (2) which is γ_{j+1}-sparse (see Definition 1) for some $1 \leq \gamma_{j+1} \leq k$. The value γ_{j+1} may a priori vary across versions of one object, and across application domains. All the versions $\mathbf{x}_1, \ldots, \mathbf{x}_L$ need protection from node failures, and are archived using a linear erasure code (see (1)).

2.1 Object Encoding

We describe a generic *differential* encoding (called **Step** $j+1$) suited for efficient archival of versioned data, which exploits the sparsity of the updates, when $\gamma_{j+1} < \frac{k}{2}$, to reduce the storage-overheads of archiving all the versions reliably. We assume that one storage node is in possession of two versions, say $\mathbf{x}_j$ and $\mathbf{x}_{j+1}$ of one data object, $j = 1, \ldots, L-1$. The corresponding implementation is discussed in Subsect. 2.2.

Step $j+1$**.** For the two versions $\mathbf{x}_j$ and $\mathbf{x}_{j+1}$, the difference vector $\mathbf{z}_{j+1} = \mathbf{x}_{j+1} - \mathbf{x}_j$ and the corresponding sparsity level γ_{j+1} are computed. If $\gamma_{j+1} \geq \frac{k}{2}$, the object $\mathbf{z}_{j+1}$ is encoded as $\mathbf{c}_{j+1} = \mathbf{G}\mathbf{z}_{j+1}$. On the other hand, if $\gamma_{j+1} < \frac{k}{2}$, then $\mathbf{z}_{j+1}$ is first compressed (see (3)) as

$$\mathbf{z}'_{j+1} = \Phi_{\gamma_{j+1}} \mathbf{z}_{j+1},$$

where $\Phi_{\gamma_{j+1}} \in \mathbb{F}_q^{2\gamma_{j+1} \times k}$ is a measurement matrix such that any $2\gamma_{j+1}$ of its columns are linearly independent (see Proposition 1). Subsequently, $\mathbf{z}'_{j+1}$ is encoded as

$$\mathbf{c}_{j+1} = \mathbf{G}_{\gamma_{j+1}} \mathbf{z}'_{j+1},$$

where $\mathbf{G}_{\gamma_{j+1}} \in \mathbb{F}_q^{n_{\gamma_{j+1}} \times 2\gamma_{j+1}}$ is the generator matrix of an $(n_{\gamma_{j+1}}, 2\gamma_{j+1})$ erasure code with storage-overhead κ. The components of $\mathbf{c}_{j+1}$ are distributed across a set $\mathcal{N}_{j+1}$ of $n_{\gamma_{j+1}}$ nodes, whose choice is discussed in Subsect. 2.2.

Since γ_{j+1} is random, a total of $\lceil \frac{k}{2} \rceil$ erasure codes denoted by

$$\mathcal{G} = \{\mathbf{G}, \mathbf{G}_1, \ldots, \mathbf{G}_{\lceil \frac{k}{2} \rceil - 1}\},$$

and a total of $\lceil \frac{k}{2} \rceil - 1$ measurement matrices denoted by $\Sigma = \{\Phi_1, \Phi_2, \ldots, \Phi_{\lceil \frac{k}{2} \rceil - 1}\}$ have to be designed a priori. The erasure codes may be taken systematic and/or MDS (that is, such that any $n-k$ failure patterns are tolerated), our scheme works irrespective of these choices. This encoding strategy implies one extra matrix multiplication whenever a sparse difference vector is obtained.

We give a toy example to illustrate the computations.

procedure ENCODE($\mathcal{X}, \mathcal{G}, \Sigma$)
  FOR $0 \leq j \leq L-1$
    IF $j = 0$
      return $\mathbf{c}_1 = \mathbf{G}\mathbf{x}_1$;
    ELSE (*This part summarizes Step* $j+1$ *in the text*)
      Compute $\mathbf{z}_{j+1} = \mathbf{x}_{j+1} - \mathbf{x}_j$;
      Compute γ_{j+1};
      IF $\gamma_{j+1} \geq \frac{k}{2}$
        return $\mathbf{c}_{j+1} = \mathbf{G}\mathbf{z}_{j+1}$;
      ELSE
        Compress $\mathbf{z}_{j+1}$ as $\mathbf{z}'_{j+1} = \Phi_{\gamma_{j+1}}\mathbf{z}_{j+1}$;
        return $\mathbf{c}_{j+1} = \mathbf{G}_{\gamma_{j+1}}\mathbf{z}_{j+1}$;
      END IF
    END IF
  END FOR
end procedure

Fig. 4. Encoding procedure for DEC

Example 1. Take $k = 4$, suppose that the digital content is written in binary as (100110010010) and that the linear code used for storage is a $(6, 4)$ code over $\mathbb{F}_8$. To create the first data object $\mathbf{x}_1$, cut the digital content into $k = 4$ chunks 100, 110, 010, 010, so that $\mathbf{x}_1$ is written over $\mathbb{F}_8$ as $\mathbf{x}_1 = (1, 1+w, w, w)$ where w is the generator of $\mathbb{F}_8^*$, satisfying $w^3 = w+1$. The next version of the digital content is created, say (10011011001). Similarly $\mathbf{x}_2$ becomes $\mathbf{x}_2 = (1, 1+w, 1+w, w)$, and the difference vector $\mathbf{z}_2$ is given by $\mathbf{z}_2 = \mathbf{x}_2 - \mathbf{x}_1 = (0, 0, 1, 0)$, with $\gamma_2 = 1 < k/2$. Apply a measurement matrix $\Phi_{\gamma_2} = \Phi_1$ to compress $\mathbf{z}_2$:

$$\Phi_1\mathbf{z}_2 = \begin{bmatrix} 1 & 0 & w & w+1 \\ 0 & 1 & w+1 & w \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} = \begin{bmatrix} w \\ w+1 \end{bmatrix} = \mathbf{z}'_2.$$

Note that every two columns of Φ_1 are linearly independent (see Proposition 1), thus allowing the compressed vector to be recovered. Encode $\mathbf{z}'_2$ using a single parity check code:

$$\mathbf{c}_2 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} w \\ w+1 \end{bmatrix} = \begin{bmatrix} w \\ w+1 \\ 1 \end{bmatrix}.$$

2.2 Implementation and Placement

Caching. To store $\mathbf{x}_{j+1}$ for $j \geq 1$, the proposed scheme requires the calculation of differences between the existing version $\mathbf{x}_j$ and the new version $\mathbf{x}_{j+1}$ in (2). However, it does not store $\mathbf{x}_j$, but $\mathbf{x}_1$ together with $\mathbf{z}_2, \ldots, \mathbf{z}_j$. Reconstructing $\mathbf{x}_j$ before computing the difference and encoding the new difference is expensive

in terms of I/O operations, network bandwidth, latency as well as computations. A practical remedy is thus to cache a full copy of the latest version $\mathbf{x}_j$, until a new version $\mathbf{x}_{j+1}$ arrives. This also helps in improving the response time and overheads of data read operations in general, and thus disentangles the system performance from the storage efficient resilient storage of all the versions.

Considering caching as a practical method, an algorithm summarizes the differential erasure coding (DEC) procedure in Fig. 4. The input and the output of the algorithm are $\mathcal{X} = \{\mathbf{x}_j \in \mathbb{F}_q^k,\ 1 \leq j \leq L\}$ and $\{\mathbf{c}_j,\ 1 \leq j \leq L\}$, respectively.

Placement Consideration. The choice of the sets $\mathcal{N}_{j+1}$, $j = 0, \ldots, L-1$ of nodes over which the different versions are stored needs a closer introspection. Since $\mathbf{x}_1$ together with $\mathbf{z}_2, \ldots, \mathbf{z}_j$ are needed to recover $\mathbf{x}_j$ (see also Subsect. 2.4), if $\mathbf{x}_1$ is lost, $\mathbf{x}_j$ cannot be recovered, and thus there is no gain in fault tolerance by storing $\mathbf{x}_j$ in a different set of nodes than $\mathcal{N}_1$. Furthermore, since $n_{\gamma_j} < n$, codewords $\mathbf{c}_i$s may have different resilience to failures. The dependency of $\mathbf{x}_j$ on previous versions suggests that the fault-tolerance of subsequent versions are determined by the worst fault-tolerance achieved among $\mathbf{c}_i$s for $i < j$.

Example 2. We continue Example 1, where $\mathbf{x}_1$ is encoded into $\mathbf{c}_1 = (c_{11}, \ldots, c_{16})$ using a $(6, 4)$ MDS code. Allocate c_{1i} to N_i, that is use the set $\mathcal{N}_1 = \{N_1, \ldots, N_6\}$ of nodes. Store $\mathbf{c}_2$ in $\mathcal{N}_2 = \{N_1, N_2, N_3\} \subset \mathcal{N}_1$ for collocated placement, and in $\mathcal{N}_2 = \{N_1', N_2', N_3'\}$, $\mathcal{N}_2 \cap \mathcal{N}_1 = \emptyset$ for distributed placement. Let p be the probability that a node fails, and failures are assumed independent. We compute the probability to recover both $\mathbf{x}_1$ and $\mathbf{x}_2$ in case of node failures (known as *static resilience*) for both distributed and collocated strategies.

For distributed placement, the set of error events for losing $\mathbf{x}_1$ is $\mathcal{E}_1 = \{3 \text{ or more nodes fail in } \mathcal{N}_1\}$. Hence, the probability $\text{Prob}(\mathcal{E}_1)$ of losing $\mathbf{x}_1$ is given by

$$p^6 + C_5^6 p^5(1-p) + C_4^6 p^4(1-p)^2 + C_3^6 p^3(1-p)^3, \tag{4}$$

where C_r^m denotes the m choose r operation. The set of error events for losing $\mathbf{z}_2$ stored with a (3,2) MDS code is $\mathcal{E}_2 = \{2 \text{ or } 3 \text{ nodes fail in } \mathcal{N}_2\}$. Thus, $\mathbf{z}_2$ is lost with probability

$$\text{Prob}(\mathcal{E}_2) = p^3 + C_2^3 p^2(1-p). \tag{5}$$

From (4) and (5), the probability of retaining both versions is

$$\text{Prob}_d(\mathbf{x}_1, \mathbf{x}_2) \triangleq (1 - \text{Prob}(\mathcal{E}_1))(1 - \text{Prob}(\mathcal{E}_2)). \tag{6}$$

The set of error events for losing $\mathbf{x}_1$ or $\mathbf{z}_2$ is

$$\mathcal{E}_1 \cup \mathcal{E}_2 = \{3 \text{ or more nodes fail}\} \cup \{\text{specific 2 nodes failure}\}$$

for collocated placement. Out of C_2^6 possible 2 node failure patterns, 3 patterns contribute to the loss of the object $\mathbf{z}_2$. Therefore, $\text{Prob}(\mathcal{E}_1 \cup \mathcal{E}_2)$ is

$$p^6 + C_5^6 p^5(1-p) + C_4^6 p^4(1-p)^2 + C_3^6 p^3(1-p)^3 + 3p^2(1-p)^4$$

from which, the probability of retaining both the versions is

$$\text{Prob}_c(\mathbf{x}_1, \mathbf{x}_2) \triangleq 1 - \text{Prob}(\mathcal{E}_1 \cup \mathcal{E}_2). \tag{7}$$

In Fig. 5, we compare (6) and (7) for different values of p from 0.001 to 0.05. The plot shows that collocated allocation results in better resilience than the distributed case.

Optimized Step $j+1$. Based on these insights, a practical change of **Step** j is: if $\gamma_{j+1} \geq \frac{k}{2}$, $\mathbf{z}_{j+1}$ is discarded and $\mathbf{x}_{j+1}$ is encoded as $\mathbf{c}_{j+1} = \mathbf{G}\mathbf{x}_{j+1}$, to ensure that a whole version is again encoded. Since many contiguous sparse versions may be created, we put as a heuristic an iteration threshold ι, after which even if all differences from one version to another stay very sparse, a whole version is used for coding and storage.

2.3 On the Storage-Overhead

Since employed erasure codes depend on the sparsity level, the storage-overhead of the above differential encoding improves upon that of encoding different versions independently. The average gains in storage-overhead are discussed in Subsect. 3.3. Formally, the total storage size till the l-th version is

$$\delta(\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_l) = n + \sum_{j=2}^{l} \min(2\kappa\gamma_j, n) \leq ln,$$

for $2 \leq l \leq L$. The storage-overhead for the **Optimized Step** $j+1$ is the same as that of **Step** $j+1$ since for $\gamma_{j+1} \geq \frac{k}{2}$, the coded objects $\mathbf{G}\mathbf{x}_{j+1}$ and $\mathbf{G}\mathbf{z}_{j+1}$ have the same size.

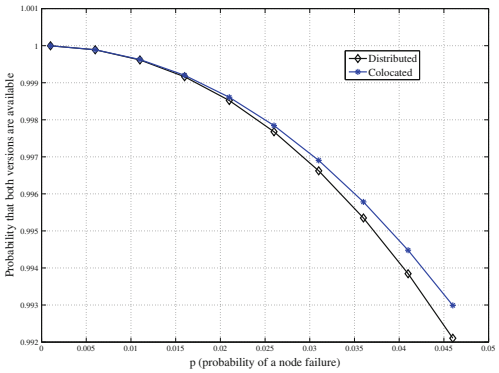

Fig. 5. Placement consideration: comparing probability that both versions are available

2.4 Object Retrieval

Suppose that L versions of a data object are archived using **Step** $j+1$, $j \leq L-1$ and the user needs to retrieve some $\mathbf{x}_l$, $1 < l \leq L$. Assuming that there are enough encoded blocks for each $\mathbf{c}_i$ $(i \leq l)$ available, relevant nodes in the sets $\mathcal{N}_1, \ldots, \mathcal{N}_l$ are accessed to fetch and decode the $\mathbf{c}_i$ to obtain $\mathbf{x}_1$, and the $l-1$ compressed differences $\mathbf{z}'_2, \mathbf{z}'_3, \ldots, \mathbf{z}'_l$. See Subsect. 2.2 for a discussion on placement and an illustration that reusing the same set of nodes gives the best availability with MDS codes, hence bounding the number of accessed nodes by $|\mathcal{N}_1|$. All compressed differences sharing the same sparsity can be added first, and then decompressed, since

$$\sum_{i \in J_\gamma} \mathbf{z}'_i = \Phi_\gamma \sum_{i \in J_\gamma} \mathbf{z}_i$$

for $J_\gamma = \{j | \gamma_j = \gamma\}$. The cost of recovering $\sum_{i \in J_\gamma} \mathbf{z}_i$ is only one decompression instead of $|J_\gamma|$, with which $\mathbf{x}_l$ is given by

$$\mathbf{x}_l = \mathbf{x}_1 + \sum_{j=2}^{l} \mathbf{z}_j.$$

A minimum of k I/O reads is needed to retrieve $\mathbf{x}_1$. For $\mathbf{z}_j$ $(2 \leq j \leq l)$, the number of I/O reads may be lower than k, depending on the update sparsity. If $\gamma_j < \frac{k}{2}$, then $\mathbf{z}'_j$ is retrieved with $2\gamma_j$ I/O reads, while if $\gamma_j \geq \frac{k}{2}$, then $\mathbf{z}_j$ is recovered with k I/O reads, so that $\min(2\gamma_j, k)$ I/O reads are needed for $\mathbf{z}_j$. The total number of I/O reads to retrieve $\mathbf{x}_l$ is

$$\eta(\mathbf{x}_l) = k + \sum_{j=2}^{l} \min(2\gamma_j, k) \tag{8}$$

and so is the total number of I/O reads to retrieve the first l versions: $\eta(\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_l) = \eta(\mathbf{x}_l)$.

To retrieve $\mathbf{x}_l$ for $1 \leq l \leq L$, when archival was done using **Optimized Step** $j+1$, $j \leq L-1$, look for the most recent version $\mathbf{x}_{l'}$ such that $l' \leq l$ and $\gamma_{l'} \geq \frac{k}{2}$. Then, using $\{\mathbf{x}_{l'}, \mathbf{z}_{l'+1}, \ldots, \mathbf{z}_l\}$, the object $\mathbf{x}_l$ is reconstructed as $\mathbf{x}_l = \mathbf{x}_{l'} + \sum_{j=l'+1}^{l} \mathbf{z}_j$. Hence, the total number of I/O reads is

$$\eta(\mathbf{x}_l) = k + \sum_{j=l'+1}^{l} \min(2\gamma_j, k). \tag{9}$$

The number of I/O reads to retrieve the first l versions is the same as for **Step** $j+1$.

Example 3. Assume that $L = 20$ versions of an object of size $k = 10$ are differentially encoded, with sparsity profile

$$\{\gamma_j,\ 2 \leq j \leq L\} = \{3, 8, 3, 6, 7, 9, 10, 6, 2, 2, 3, 9, 3, 9, 3, 10, 4, 2, 3\}.$$

The storage pattern is $\{\mathbf{x}_1, \mathbf{z}_2, \mathbf{z}_3, \ldots, \mathbf{z}_{20}\}$. Assuming $\mathbf{x}_1$ is not sparse, the I/O read numbers to access $\{\mathbf{x}_1, \mathbf{z}_2, \mathbf{z}_3, \ldots, \mathbf{z}_{20}\}$ are

$$\{10, 6, 10, 6, 10, 10, 10, 10, 10, 4, 4, 6, 10, 6, 10, 6, 10, 8, 4, 6\}.$$

The total I/O reads to recover all the 20 versions is 156 (instead of 200 for the non-differential method). The total storage space for all the 20 versions assuming a storage-overhead of 2 is 312 (instead of 400 otherwise). The I/O read numbers to recover $\{\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3, \ldots, \mathbf{x}_{20}\}$ are

$$\{10, 16, 26, 32, 42, 52, 62, 72, 82, 86, 90, 96, 106, 112, 122, 128, 138, 146, 150, 156\},$$

while for the optimized step, we get $\{10, 16, 10, 16, 10, 10, 10, 10, 10, 14, 18, 24, 10, 16, 10, 16, 10, 18, 22, 28\}$.

3 Reverse Differential Erasure Coding

In Table 1, we summarize the total storage size and the number of I/O reads required by the (forward) differential method. If some γ_j, $1 \leq j \leq l$, are smaller than $\frac{k}{2}$, then the number of I/O reads for joint retrieval of all the versions $\{\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_l\}$ is lower than that of the traditional method. However, this advantage comes at the cost of higher number of I/O reads for accessing the l-th version $\mathbf{x}_l$ alone. Therefore, for applications where the latest archived versions are more frequently accessed than the joint versions, the overhead for reading the latest version dominates the advantage of reading multiple versions. For such applications, we apply a variant of the differential method called the reverse DEC, wherein the order of storing the difference vectors is reversed [6].

Table 1. I/O access metrics for the traditional and the differential schemes to store $\{\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_l\}$

Parameter	Traditional	Forward differential	Reverse differential
I/O reads to read the l-th version	k	$k + \sum_{j=2}^{l} \min(2\gamma_j, k)$	k
I/O reads to read the first l-th versions	lk	$k + \sum_{j=2}^{l} \min(2\gamma_j, k)$	$k + \sum_{j=2}^{l} \min(2\gamma_j, k)$
Number of Encoding operations	1 (on the version)	1 (on the latest version)	2 (on the latest and the preceding version)
Total Storage Size till the l-th version	ln	$n + \sum_{j=2}^{l} \min(2\kappa\gamma_j, n)$	$n + \sum_{j=2}^{l} \min(2\kappa\gamma_j, n)$

3.1 Object Encoding

As in Subsect. 2.1, we assume that one node stores the latest version $\mathbf{x}_j$ and the new version $\mathbf{x}_{j+1}$ of a data object. Since $\mathbf{x}_j$ is readily obtained, caching is less critical here.

Step $j+1$**.** Compute the difference vector $\mathbf{z}_{j+1} = \mathbf{x}_{j+1} - \mathbf{x}_j$ and its sparsity level γ_{j+1}. The object $\mathbf{x}_{j+1}$ is encoded as $\mathbf{c}_{j+1} = \mathbf{G}\mathbf{x}_{j+1}$ and stored in $\mathcal{N}_{j+1}$. Furthermore, if $\gamma_{j+1} < \frac{k}{2}$, then $\mathbf{z}_{j+1}$ is first compressed as $\mathbf{z}'_{j+1} = \Phi_{\gamma_{j+1}}\mathbf{z}_{j+1}$, and then encoded as $\mathbf{c} = \mathbf{G}_{\gamma_{j+1}}\mathbf{z}'_{j+1}$, where $\mathbf{G}_{\gamma_{j+1}}$ is the generator matrix of an $(n_{\gamma_{j+1}}, 2\gamma_{j+1})$ erasure code. Finally, the preceding version $\mathbf{c}_j$ is overwritten as $\mathbf{c}_j = \mathbf{c}$.

A key feature is that in addition to encoding the latest version $\mathbf{x}_{j+1}$, the preceding version is also re-encoded depending on the sparsity level γ_{j+1}, resulting in two encoding operations (instead of one for the method in Subsect. 2.1).

A summary of the encoding is provided in Fig. 6. The storage-overhead for this method is the same as the one in Sect. 2. The considerations on data placement and static resilience of $\mathbf{c}_j$ in the set $\mathcal{N}_j$ of nodes are analogous as well, and an optimized version is obtained similarly as for the forward differential encoding.

3.2 Object Retrieval

Suppose that l versions of a data object have been archived, and the user needs to retrieve the latest version $\mathbf{x}_l$. In the reverse DEC, unlike Subsect. 2.1, the latest version $\mathbf{x}_l$ is encoded as $\mathbf{G}\mathbf{x}_l$. Hence, the user must access a minimum of k nodes from the set $\mathcal{N}_l$ to recover $\mathbf{x}_l$. To retrieve all the l versions $\{\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_l\}$, the user accesses the nodes in the sets $\mathcal{N}_1, \mathcal{N}_2, \ldots, \mathcal{N}_l$ to retrieve $\mathbf{z}'_2, \mathbf{z}'_3, \ldots, \mathbf{z}'_l, \mathbf{x}_l$,

```
1: procedure ENCODE(𝒳, 𝒢, Σ)
2:     FOR 0 ≤ j ≤ L − 1
3:         IF j = 0
4:             return c_1 = Gx_1;
5:         ELSE (This part summarizes Step j + 1 in the text)
6:             c_{j+1} = Gx_{j+1};
7:             Compute z_{j+1} = x_{j+1} − x_j;
8:             Compute γ_{j+1};
9:             IF γ_{j+1} < k/2
10:                Compress z_{j+1} as z'_{j+1} = Φ_{γ_{j+1}} z_{j+1};
11:                return c_j = G_{γ_{j+1}} z_{j+1};
12:            END IF
13:        END IF
14:    END FOR
15: end procedure
```

Fig. 6. Encoding procedure for the reverse DEC

respectively. The objects $\mathbf{z}_2, \mathbf{z}_3, \ldots, \mathbf{z}_l$ are recovered from $\mathbf{z}'_2, \mathbf{z}'_3, \ldots, \mathbf{z}'_l$, respectively through a sparse-reconstruction procedure, and $\mathbf{x}_j$, $1 \leq j \leq l-1$, is recursively reconstructed as

$$\mathbf{x}_j = \mathbf{x}_l - \left(\sum_{t=j}^{l} \mathbf{z}_t \right).$$

It is clear that a total of $k + \sum_{j=2}^{l} \min(2\gamma_j, k)$ reads are needed for accessing all the l versions and only k reads for the latest version. The performance metrics of the reverse DEC scheme are also summarized in Table 1 (the last column).

Example 4. For the sparsity profile of Example 3, the storage pattern using reverse DEC is $\{\mathbf{z}_2, \mathbf{z}_3, \ldots, \mathbf{z}_{20}, \mathbf{x}_{20}\}$. The I/O read numbers to access $\{\mathbf{z}_2, \ldots, \mathbf{x}_{20}\}$ are $\{6, 10, 6, 10, 10, 10, 10, 10, 4, 4, 6, 10, 6, 10, 6, 10, 8, 4, 6, 10\}$. The total storage size and the I/O reads to recover all the 20 versions are the same as that of the forward differential method. The I/O numbers to recover $\{\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3, \ldots, \mathbf{x}_{20}\}$ are $\{156, 150, 144, 134, 124, 114, 104, 94, 84, 80, 76, 70, 60, 54, 44, 38, 28, 20, 16, 10\}$. Note that I/O number to access the latest version (in this case 20th version) is lower than that of the forward differential scheme. For the optimized step, the corresponding I/O numbers are $\{16, 10, 16, 10, 10, 10, 10, 10, 24, 20, 16, 10, 16, 10, 16, 10, 28, 20, 16, 10\}$.

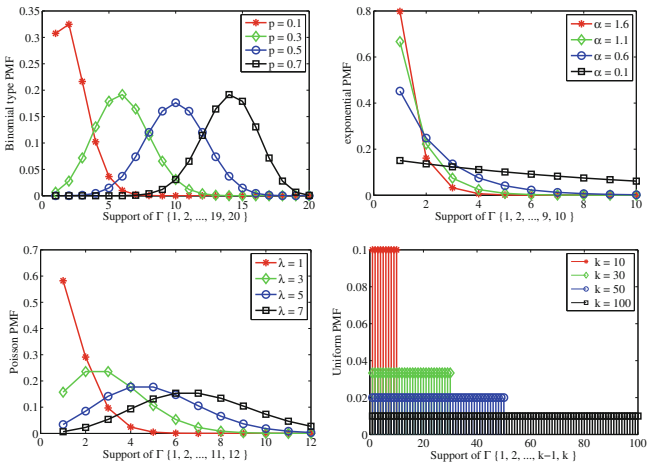

Fig. 7. From top left, clock-wise: Binomial type PMF in p (for k = 20), Truncated exponential PMF in α (for k = 10), Truncated Poisson PMF in λ (for k = 12) and the uniform PMF for different object lengths k. The x-axis of these plots represent the support $\{1, 2, \ldots, k\}$ of the random variable Γ.

3.3 Exploring DEC Benefits with Synthetic Workloads

In this section, we quantify the storage savings offered by the DEC against update patterns that are characterized by in-place alterations. For this study, the update model follows (2), and the in-place alterations are generated from synthetic workloads from a wide-rage of distributions. This exercise showcases the best-case storage savings of DEC as fewer in-place alterations guarantee corresponding sparsity levels in the difference objects, unlike the case of fewer insertions and deletions that totally disturb the sparsity profile.

We present experimental results on the storage size and the number of I/O reads for the different differential encoding schemes. We assume that $\{\Gamma_j,\ 2 \leq j \leq L\}$ is a set of random variables and its realizations $\{\gamma_j,\ 2 \leq j \leq L\}$ are known. First we consider a version-control system with $L = 2$, which is the worst-case choice of L as more versions could reveal more storage savings. This setting both (i) serves as a proof of concept, and (ii) already shows the storage savings for this simple case. Later, we also present experimental results for a setup with $L > 2$ versions.

System with $L = 2$ Versions. For $L = 2$, there is one random variable denoted henceforth as Γ, with realization γ. Since Γ is a discrete random variable with finite support, we test the following finite support distributions for our experimental results on the average number of I/O reads for the two versions and the average storage size.

Binomial Type PMF: This is a variation of the standard Binomial distribution given by

$$\mathrm{P}_{\Gamma}(\gamma) = c\frac{k!}{\gamma!(k-\gamma)!}p^{\gamma}(1-p)^{k-\gamma},\ \gamma = 1, 2, \ldots, k, \tag{10}$$

where $c = \frac{1}{1-(1-p)^k}$ is the normalizing constant. The change is necessary since $\gamma = 0$ is not a valid event.

Truncated Exponential PMF: This is a finite support version of the exponential distribution in parameter $\alpha > 0$:

$$\mathrm{P}_{\Gamma}(\gamma) = ce^{-\alpha\gamma}. \tag{11}$$

The constant c is chosen such that $\sum_{\gamma=1}^{k} \mathrm{P}_{\Gamma}(\gamma) = 1$.

Truncated Poisson PMF: This is a finite support version of the Poisson distribution in parameter λ given by

$$\mathrm{P}_{\Gamma}(\gamma) = c\frac{\lambda^{\gamma}e^{-\lambda}}{\gamma!}, \tag{12}$$

where the constant c is chosen such that $\sum_{\gamma=1}^{k} \mathrm{P}_{\Gamma}(\gamma) = 1$

Uniform PMF: This is the standard uniform distribution:

$$\mathrm{P}_{\Gamma}(\gamma) = \frac{1}{k}. \tag{13}$$

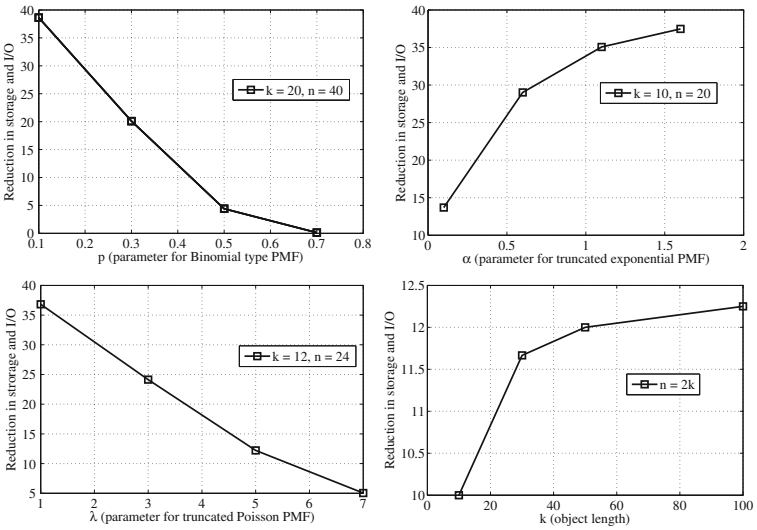

Fig. 8. Average percentage reduction in the I/O reads and storage size for PMFs in Fig. 7 when $L = 2$. The experimental results are presented in the same order as that of the PMFs in Fig. 7.

In Fig. 7, we plot the PMFs in (10), (11), (12) and (13) for various parameters. These PMFs are chosen to represent a wide range of real-world data update scenarios, in the absence of any standard benchmarking dataset (see [16]). The truncated exponential PMFs generate thick concentration for lower sparsity levels, yielding best cases for the differential encodings. The uniform distributions illustrate the benefits of the proposed methods for update patterns with no bias on sparse values. The Binomial distributions provide narrow and bell shaped mass functions concentrated around different sparsity levels. The Poisson PMFs model sparse updates spread over the entire support and concentrated around the center.

For a given PMF $\mathrm{P}_{\Gamma}(\gamma)$, the average storage size for storing the first two versions is $\mathbb{E}[\delta(\mathbf{x}_1, \mathbf{x}_2)] = n + \sum_{\gamma=1}^{k} \mathrm{P}_{\Gamma}(\gamma)\min(2\gamma\kappa, n)$ where $n = \kappa k$. Similarly, the average number of I/O reads to access the first two versions is $\mathbb{E}[\eta(\mathbf{x}_1, \mathbf{x}_2)] = k + \sum_{\gamma=1}^{k} \mathrm{P}_{\Gamma}(\gamma)\min(2\gamma, k)$. When compared to the non-differential method, the average percentage reduction in the I/O reads and the average percentage reduction in the storage size are respectively computed as

$$\frac{2k - \mathbb{E}[\eta(\mathbf{x}_1, \mathbf{x}_2)]}{2k} \times 100 \text{ and } \frac{2n - \mathbb{E}[\delta(\mathbf{x}_1, \mathbf{x}_2)]}{2n} \times 100. \tag{14}$$

Since $\delta(\mathbf{x}_1, \mathbf{x}_2) = \kappa\eta(\mathbf{x}_1, \mathbf{x}_2)$ and κ is a constant, the numbers in (14) are identical. In Fig. 8, we plot the percentage reduction in the above quantities for the PMFs displayed in Fig. 7. The plots show a significant reduction in the I/O reads (and the storage size) when the distributions are skewed towards smaller γ. However, as expected, the reduction is marginal otherwise. For uniform distribution

on Γ, the plot shows that the advantage with the differential technique saturates for large values of k.

We have discussed how the differential technique reduces the storage space at the cost of increased number of I/O reads for the latest version (here the 2nd version) when compared to the non-differential method. For the basic differential encoding, the average number of I/O reads to retrieve the 2nd version is $\mathbb{E}[\eta(\mathbf{x}_2)] = \mathbb{E}[\eta(\mathbf{x}_1, \mathbf{x}_2)]$. However, for the optimized encoding, $\mathbb{E}[\eta(\mathbf{x}_2)] = \sum_{\gamma=1}^{k} \mathrm{P}_{\Gamma}(\gamma) f(\gamma)$ where $f(\gamma) = k + 2\gamma$ when $\gamma < \frac{k}{2}$, and $f(\gamma) = k$, otherwise. When compared to the non-differential method, we compute the average percentage increase in the I/O reads for retrieving the 2nd version for both the basic and the optimized methods. Numbers for

$$\frac{\mathbb{E}[\eta(\mathbf{x}_2)] - k}{k} \times 100, \tag{15}$$

are shown in Fig. 9, which shows that the optimized method reduces the excess number of I/O reads for the 2nd version.

Experimental Results for $L > 2$. We present the average reduction in the total storage size for a differential system with $L = 10$, assuming identical PMFs on the sparsity levels for every version, i.e., $\mathrm{P}_{\Gamma_j}(\gamma_j) = \mathrm{P}_{\Gamma}(\gamma)$ for each $2 \leq j \leq 10$. The average percentage reduction in the total storage size and total I/O reads number are computed similarly to (14), and are illustrated in Fig. 10. The plots show further increase in storage savings compared to $L = 2$ case. In reality, the PMFs across different versions may be different and possibly correlated. These results are thus only indicative of the saving magnitude for storing many versions differentially.

To get better insights for $L > 2$, in Fig. 11, we plot the I/O numbers of Examples 3 and 4 for $L = 20$. More than 20% storage space is saved with respect to the non-differential scheme, for only slightly higher I/O for the optimized DEC.

4 Two-Level Differential Erasure Coding

The differential encoding (both forward and the reverse DEC) exploits the sparse nature of the updates to reduce the storage size and the number of I/O reads. Such advantages stem from the application of $\lceil \frac{k}{2} \rceil$ erasure codes matching the different levels of sparsity ($\lceil \frac{k}{2} \rceil - 1$ erasure codes for each $\gamma < \frac{k}{2}$ and one for $\gamma \geq \frac{k}{2}$). If k is large, then the system needs a large number of erasure codes, resulting in an impractical strategy. In this section, we employ only two erasure codes, termed *two-level differential erasure coding*, for the sake of easier implementation, and refer to the earlier differential schemes in Subsects. 2 and 3 as $\frac{k}{2}$-level DEC schemes. We need the following ingredients for the two-level DEC scheme:

(1) An (n, k) erasure code with generator matrix $\mathbf{G} \in \mathbb{F}_q^{n \times k}$ to store the original data object.

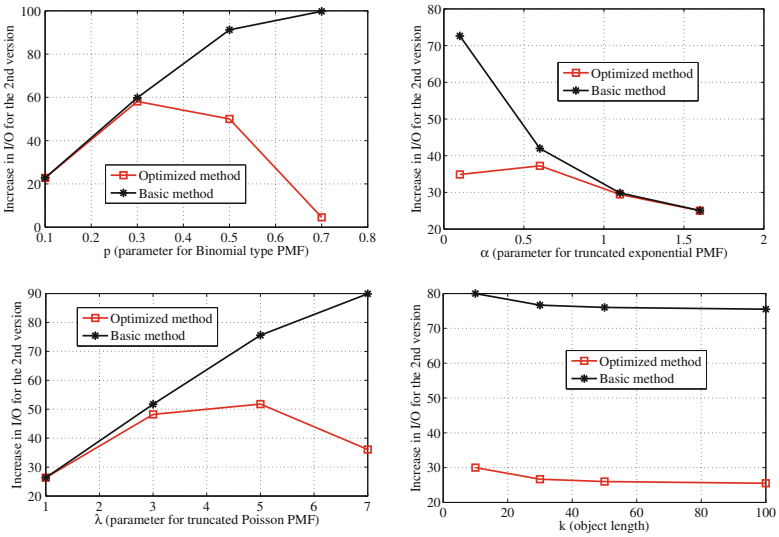

Fig. 9. Average percentage increase in the I/O reads to retrieve the 2nd version for the PMFs (in the same order) in Fig. 7 when $L = 2$. The corresponding values of n and k are same as that of Fig. 8.

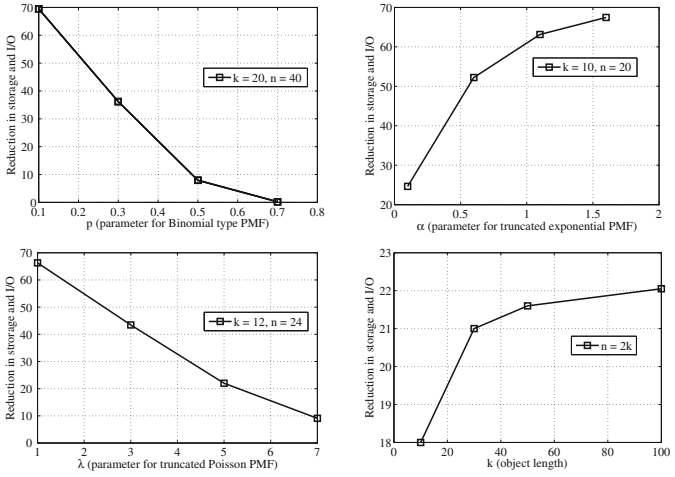

Fig. 10. Average percentage reduction in the I/O reads and total storage size for PMFs in Fig. 7 when $L = 10$. The experimental results are presented in the same order as that of the PMFs in Fig. 7. Identical PMFs are used for the random variable $\{\Gamma_j, 2 \leq j \leq 10\}$ to obtain the results.

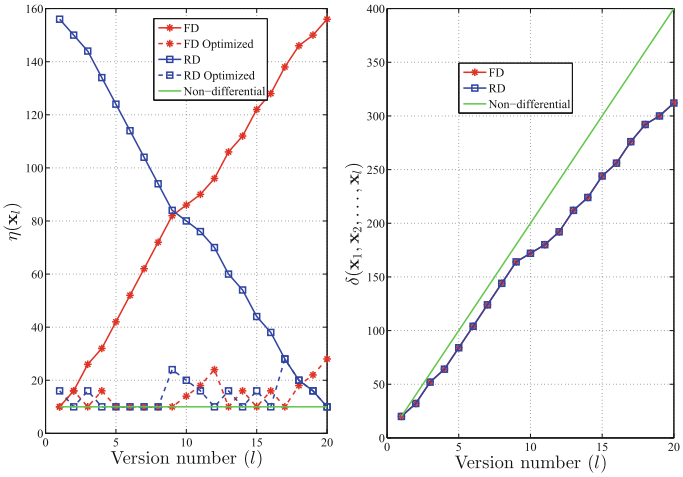

Fig. 11. I/O and storage for Examples 3 and 4. The left plots provide the number of I/O reads to retrieve only the l-th version for $1 \leq l \leq 20$. The right plots show the total storage size till the l-th version for $1 \leq l \leq 20$. Results are for forward and reverse differential methods, with basic and optimized encoding.

(2) A measurement matrix $\Phi_T \in \mathbb{F}_q^{2T \times k}$ to compress sparse updates, where $T \in \{1, 2, \ldots, \lfloor \frac{k}{2} \rfloor\}$ is a chosen threshold.
(3) An $(n_T, 2T)$ erasure code with generator matrix $\mathbf{G}_T \in \mathbb{F}_q^{n_T \times 2T}$ to store the compressed data object. The number n_T is chosen such that $\kappa \triangleq \frac{n}{k} = \frac{n_T}{2T}$.

We discuss only the two-level *forward* DEC scheme. The two-level *reverse* DEC scheme is a straightforward variation.

4.1 Object Encoding

The key point of this encoding is that the number of erasure codes (and the corresponding measurement matrices) to store the γ-sparse vectors for $1 \leq \gamma < \frac{k}{2}$ is reduced from $\lceil \frac{k}{2} \rceil - 1$ to 1. Thus, based on the sparsity level, the update vector is either compressed and then archived, or archived as it. Formally:

Step $j + 1$**.** Once the version $\mathbf{x}_{j+1}$ is created, using $\mathbf{x}_j$ in the cache, the difference vector $\mathbf{z}_{j+1} = \mathbf{x}_{j+1} - \mathbf{x}_j$ and the corresponding sparsity level γ_{j+1} are computed. If $\gamma_{j+1} > T$, the object $\mathbf{z}_{j+1}$ is encoded as $\mathbf{c}_{j+1} = \mathbf{G}\mathbf{z}_{j+1}$, else $\mathbf{z}_{j+1}$ is first compressed (see (3)) as $\mathbf{z}'_{j+1} = \Phi_T \mathbf{z}_{j+1}$, where the measurement matrix $\Phi_T \in \mathbb{F}_q^{2T \times k}$ is such that any $2T$ of its columns are linearly independent (see Proposition 1). Then, $\mathbf{z}'_{j+1}$ is encoded as $\mathbf{c}_{j+1} = \mathbf{G}_T \mathbf{z}'_{j+1}$, where $\mathbf{G}_T \in \mathbb{F}_q^{n_T \times 2T}$ is the generator matrix of an $(n_T, 2T)$ erasure code. The components of $\mathbf{c}_{j+1}$ are stored across the set $\mathcal{N}_{j+1}$ of nodes.

A summary of the encoding method is provided in Fig. 12.

```
1: procedure ENCODE(𝒳, G, G_T, Φ_T)
2:     FOR 0 ≤ j ≤ L − 1
3:         IF j = 0
4:             return c_1 = Gx_1;
5:         ELSE (This part summarizes Step j + 1 in the text)
6:             Compute z_{j+1} = x_{j+1} − x_j;
7:             Compute γ_{j+1};
8:             IF γ_{j+1} > T
9:                 return c_{j+1} = Gz_{j+1};
10:            ELSE
11:                Compress z_{j+1} as z'_{j+1} = Φ_T z_{j+1};
12:                return c_{j+1} = G_T z_{j+1};
13:            END IF
14:        END IF
15:    END FOR
16: end procedure
```

Fig. 12. Encoding procedure for two-level DEC

4.2 On the Storage-Overhead

The total storage size for the two-level DEC is $\delta(\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_l) = n + \sum_{j=2}^{l} n_j$, where

$$n_j = \begin{cases} n, \text{ if } \gamma_j > T \\ \kappa 2T, \text{ otherwise.} \end{cases} \tag{16}$$

4.3 Data Retrieval

Similarly to the $\frac{k}{2}$-level DEC scheme, the object $\mathbf{x}_l$ for some $1 \leq l \leq L$ is reconstructed as $\mathbf{x}_l = \mathbf{x}_1 + \sum_{j=2}^{l} \mathbf{z}_j$, by accessing the nodes in the sets $\mathcal{N}_1, \mathcal{N}_2, \ldots, \mathcal{N}_l$. To retrieve $\mathbf{x}_1$, a minimum of k I/O reads is needed. If $\mathbf{z}_j$ is γ_j-sparse and $\gamma_j \leq T$, then $\mathbf{z}'_j$ is first retrieved with $2T$ I/O reads, second, $\mathbf{z}_j$ is decoded from $\mathbf{z}'_j$ and Φ_T through a sparse-reconstruction procedure. On the other hand, if $\gamma_j > T$, then $\mathbf{z}_j$ is recovered with k I/O reads. Overall, the total number of I/O reads for $\mathbf{x}_l$ in the differential set up is $\eta(\mathbf{x}_l) = k + \sum_{j=2}^{l} \eta_j$, where

$$\eta_j = \begin{cases} 2T, \text{ if } \gamma_j \leq T \\ k, \text{ otherwise.} \end{cases} \tag{17}$$

Similarly, the total number of I/O reads to retrieve the first l versions is also $\eta(\mathbf{x}_1, \ldots, \mathbf{x}_l) = k + \sum_{j=2}^{l} \eta_j$.

Example 5. We apply the threshold $T = 3$ to the sparsity profile in Example 3. The object $\mathbf{z}_{18}$ (with $\gamma_{18} = 4$) is then archived without compression whereas all objects with sparsity lower than or equal to 3 are compressed using a 6×10 measurement matrix. The I/O read numbers to access $\{\mathbf{x}_1, \mathbf{z}_2, \mathbf{z}_3, \ldots, \mathbf{z}_{20}\}$ are

$\{10, 6, 10, 6, 10, 10, 10, 10, 10, 6, 6, 6, 10, 6, 10, 6, 10, 10, 6, 6\}$. The total number of I/O reads to access all the versions is 164 and the corresponding storage size is 328. Thus, with just two levels of compression, the storage-overhead is more than the 5-level DEC scheme but still lower than 400.

4.4 Threshold Design Problem

For the two-level DEC, the total number of I/O reads and the storage size are random variables that are respectively given by $\eta = k + \sum_{j=2}^{L} \eta_j$, where η_j is given in (17) and $\delta = n + \sum_{j=2}^{L} n_j$, where n_j is given in (16). Note that η and δ are also dependent on the threshold T. The threshold T that minimizes the average values of η and δ is given by:

$$T_{\text{opt}} = \arg \min_{T \in \{1,2,\ldots,\lfloor \frac{k}{2} \rfloor\}} w\mathbb{E}[\delta(\mathbf{x}_1, \mathbf{x}_2)] + (1-w)\mathbb{E}[\eta(\mathbf{x}_1, \mathbf{x}_2)], \tag{18}$$

where $0 \leq w \leq 1$ is a parameter that appropriately weighs the importance of storage-overhead and I/O reads overhead, and $\mathbb{E}[\cdot]$ is the expectation operator over the random variables $\{\Gamma_2, \Gamma_3, \ldots, \Gamma_L\}$. This optimization depends on the underlying probability mass functions (PMFs) on $\{\Gamma_j\}$, so we discuss the choice of the parameter $1 \leq T \leq \lfloor \frac{k}{2} \rfloor$ in Subsect. 4.6.

4.5 Cauchy Matrices for Two-Level DEC

Suppose that $\Phi_T \in \mathbb{F}_q^{2T \times k}$ is carved from a Cauchy matrix [22]. A Cauchy matrix is such that any square submatrix is full rank [23]. Thus, there exists a $2\gamma_j \times k$ submatrix $\Phi_T(\mathcal{I}_{2\gamma_j}, :)$ of Φ_T, where $\mathcal{I}_{2\gamma_j} \subset \{1, 2, \ldots, 2T\}$ represents the indices of $2\gamma_j$ rows, for which any $2\gamma_j$ columns are linearly independent, implying that the observations $\mathbf{r} = \Phi_T(\mathcal{I}_{2\gamma_j}, :)\mathbf{z}_j$, can be retrieved from $\mathcal{N}_j$ with $2\gamma_j$ I/O reads. Also, using $\mathbf{r}$ and $\Phi_T(\mathcal{I}_{2\gamma_j}, :)$, the sparse update $\mathbf{z}_j$ can be decoded through a sparse-reconstruction procedure. Thus, the number of I/O reads to get $\mathbf{z}_j$ is reduced from $2T$ to $2\gamma_j$ when $\gamma_j \leq T$. This procedure is applicable for any $\gamma_j < T$. Therefore, a γ_j-sparse vector with $\gamma_j \leq T$ can be recovered with $2\gamma_j$ I/O reads. The total number of I/O reads for $\mathbf{x}_l$ in the two-level DEC with Cauchy matrix is finally $\eta(\mathbf{x}_l) = k + \sum_{j=2}^{l} \eta_j$, where

$$\eta_j = \begin{cases} 2\gamma_j, & \text{if } \gamma_j \leq T \\ k, & \text{otherwise.} \end{cases} \tag{19}$$

Since the number of I/O reads is potentially different compared to the case without Cauchy matrices, the threshold design problem in (18) can result in different answers for this case. We discuss this optimization problem in Subsect. 4.6.

Example 6. With Cauchy matrix for Φ_T in Example 5, the I/O numbers to access $\{\mathbf{z}_2, \mathbf{z}_3, \ldots, \mathbf{z}_{20}, \mathbf{x}_{20}\}$ are $\{10, 6, 10, 6, 10, 10, 10, 10, 10, 4, 4, 6, 10, 6, 10, 6, 10, 10, 4, 6\}$, which makes the total I/O reads 158. However, the total storage size with Cauchy matrix continues to be 328.

4.6 Code Design Exploration for Two-Level DEC with Synethetic Workloads

In this section, we explore the right choice of threshold T for the two-level DEC scheme. A wide rage of synthetic workloads for the two-level DEC help us identify update patterns where the two-level scheme could be applicable as a substitute to $\frac{k}{2}$-level DEC.

We now present simulation results to choose the threshold parameter $1 \leq T \leq \lfloor \frac{k}{2} \rfloor$ for the two-level DEC scheme in Subsect. 4.4. The optimization problem is given in (18) where

$$\mathbb{E}[\eta(\mathbf{x}_1, \mathbf{x}_2)] = k + \mathrm{P}_\Gamma(\gamma \leq T)2T + \mathrm{P}_\Gamma(\gamma > T)k,$$

$\mathbb{E}[\delta(\mathbf{x}_1, \mathbf{x}_2)] = \kappa \mathbb{E}[\eta(\mathbf{x}_1, \mathbf{x}_2)]$ and $0 \leq w \leq 1$. Since $\mathbb{E}[\delta(\mathbf{x}_1, \mathbf{x}_2)]$ and $\mathbb{E}[\eta(\mathbf{x}_1, \mathbf{x}_2)]$ are proportional, solving (18) is equivalent to solving instead

$$T_{\text{opt}} = \arg min_{1 \leq T \leq \lfloor \frac{k}{2} \rfloor} \mathbb{E}[\delta(\mathbf{x}_1, \mathbf{x}_2)]. \tag{20}$$

In Table 2, we list the values of T_{opt}, obtained via exhaustive search over $1 \leq T \leq \lfloor \frac{k}{2} \rfloor$, the average number of I/O reads, the average storage size for the optimized two-level DEC scheme and the $\frac{k}{2}$-level DEC scheme. We denote $\mathbb{E}[\eta(\mathbf{x}_1, \mathbf{x}_2)]$ and $\mathbb{E}[\delta(\mathbf{x}_1, \mathbf{x}_2)]$ by $\mathbb{E}[\eta]$ and $\mathbb{E}[\delta]$, respectively. To compute the average storage size,

Table 2. Optimal threshold value for various PMFs with $k = 10$.

Binomial: $k = 20$, for $\frac{k}{2}$-level: $\eta = 40$ and $\delta = 80$

p	T_{opt}	$\mathbb{E}[\eta]$ (2-level)	$\mathbb{E}[\delta]$ (2-level)	$\mathbb{E}[\eta]$ ($\frac{k}{2}$-level)	$\mathbb{E}[\delta]$ ($\frac{k}{2}$-level)
0.1	3	28.11	56.23	24.55	49.10
0.3	6	35.13	70.27	31.96	63.92
0.5	8	38.99	77.98	38.23	76.47
0.7	9	39.96	79.93	39.95	79.90

Truncated Exponential: $k = 10$, for $\frac{k}{2}$-level: $\eta = 20$ and $\delta = 40$

α	T_{opt}	$\mathbb{E}[\eta]$ (2-level)	$\mathbb{E}[\delta]$ (2-level)	$\mathbb{E}[\eta]$ ($\frac{k}{2}$-level)	$\mathbb{E}[\delta]$ ($\frac{k}{2}$-level)
1.6	1	13.61	27.23	12.50	25.01
1.1	1	14.66	29.32	12.98	25.97
0.6	2	15.79	31.59	14.19	28.39
0.1	2	18.27	36.55	17.26	34.52

Truncated Poisson: $k = 12$, for $\frac{k}{2}$-level: $\eta = 24$ and $\delta = 48$

λ	T_{opt}	$\mathbb{E}[\eta]$ (2-level)	$\mathbb{E}[\delta]$ (2-level)	$\mathbb{E}[\eta]$ ($\frac{k}{2}$-level)	$\mathbb{E}[\delta]$ ($\frac{k}{2}$-level)
1	2	17.01	34.03	15.16	30.32
3	3	20.22	40.45	18.20	36.41
5	4	22.24	44.49	21.06	42.13
7	4	23.29	46.58	22.79	45.58

we use $\kappa = 2$. We see that switching to just two levels of compression incurs negligible loss in the I/O reads (or storage size) when compared to the $\frac{k}{2}$-level DEC scheme. Thus the two-level DEC scheme is a practical solution to reap the benefits of the differential erasure coding strategy.

When Cauchy matrices are used for Φ_T, (18) has to be solved for both

$$\mathbb{E}[\eta(\mathbf{x}_1, \mathbf{x}_2)] = k + \sum_{\gamma=1}^{T} \mathrm{P}_\Gamma(\gamma \leq \gamma) 2\gamma + \mathrm{P}_\Gamma(\gamma > T)k$$
$$\mathbb{E}[\delta(\mathbf{x}_1, \mathbf{x}_2)] = n + \mathrm{P}_\Gamma(\gamma \leq T) 2T\kappa + \mathrm{P}_\Gamma(\gamma > T)k\kappa.$$

Unlike the non-Cauchy case, $\mathbb{E}[\eta(\mathbf{x}_1, \mathbf{x}_2)]$ and $\mathbb{E}[\delta(\mathbf{x}_1, \mathbf{x}_2)]$ are no more proportional and T_{opt} depends on w, $0 \leq w \leq 1$.

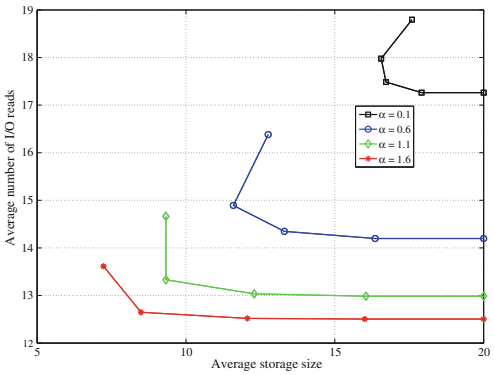

Fig. 13. Average storage size $\mathbb{E}[\delta(\mathbf{x}_1, \mathbf{x}_2)]$ versus average number of I/O reads $\mathbb{E}[\eta(\mathbf{x}_1, \mathbf{x}_2)]$, $1 \leq T \leq \lfloor \frac{k}{2} \rfloor = 5$ with truncated exponential distribution. For each curve, points from left to right tip correspond to $T = \{1, \ldots, \lfloor \frac{k}{2} \rfloor = 5\}$.

To capture the dependency on w, we study the relation between $\mathbb{E}[\eta(\mathbf{x}_1, \mathbf{x}_2)]$ and $\mathbb{E}[\delta(\mathbf{x}_1, \mathbf{x}_2)]$ for $1 \leq T \leq \lfloor \frac{k}{2} \rfloor$. In Fig. 13, we plot

$$\{(\mathbb{E}[\delta(\mathbf{x}_1, \mathbf{x}_2)], \mathbb{E}[\eta(\mathbf{x}_1, \mathbf{x}_2)]), 1 \leq T \leq \frac{k}{2}\}$$

for the exponential PMFs from Subsect. 3.3. For each curve there are $\frac{k}{2} = 5$ points corresponding to $T \in \{1, 2, \ldots, 5\}$ in that sequence from left tip to the right one. The plots indicate the value of $T_{\text{opt}}(w)$ for the two extreme values of w, i.e., $w = 0$ and $w = 1$. We further study the curve corresponding to $\alpha = 0.6$. If minimizing $\mathbb{E}[\eta(\mathbf{x}_1, \mathbf{x}_2)]$ is most important with no constraint on $\mathbb{E}[\delta(\mathbf{x}_1, \mathbf{x}_2)]$ (i.e., $w = 1$), then choose $T_{\text{opt}}(1) = \frac{k}{2}$. This option results in $\mathbb{E}[\eta(\mathbf{x}_1, \mathbf{x}_2)]$ which is as low as for the $\frac{k}{2}$-level DEC scheme. While if minimizing $\mathbb{E}[\delta(\mathbf{x}_1, \mathbf{x}_2)]$ is most important with no constraint on $\mathbb{E}[\eta(\mathbf{x}_1, \mathbf{x}_2)]$ (i.e., $w = 0$), then $T_{\text{opt}}(0) = 2$

results in $\mathbb{E}[\delta(\mathbf{x}_1, \mathbf{x}_2)]$ which is the same as for the 2-level DEC scheme with non-Cauchy matrix. For other values of w, the optimal value depends on whether $w > 0.5$. It can be found via exhaustive search over $1 \leq T \leq \lfloor \frac{k}{2} \rfloor$. In summary, using Cauchy matrix for Φ_T reduces the average number of I/O reads to that of the $\frac{k}{2}$-level DEC with just two levels of compression.

5 Practical Differential Erasure Coding

So far, we developed a theoretical framework for the DEC scheme under a *fixed object length* assumption across successive versions of the data object (see (2)). This assumption typically does not hold in practice because of insertions and deletions, which impact the length of the updated object. In this section, we explain how to control zero pads in the file structure so as to support insertions and deletions in a file, while marginally impacting the storage-overheads.

To exemplify the use of zero pads, consider storing a digital object of size 3781 units through a $(12, 8)$ erasure code of symbol size 500 units, as shown in Fig. 14. Since the object is encoded blockwise, 219 zero pads are added to extend the object size to 4000 units. The zero pads naturally absorb insertions made anywhere in the file, as long as the total size is less than 219 units, thus retaining the length of the updated version to 4000 units. However, since the zero pads are placed at the end, insertions made at the beginning of the file propagate changes across the rest of the file. The difference object is thus unlikely to exhibit sparsity. Alternatively, one could distribute zero pads across the file at different places as shown in the bottom figure of Fig. 14. Here 160 zero pads are distributed at 8 patches with each patch containing 20 zero pads. This strategy arrests propagation of changes when (small size) insertions are made either at the beginning or middle of the file.

Despite zero padding looking like a natural way to handle insertions, it is already clear from this example that the optimization of the size and placements of zero pads is not immediate. We defer this analysis to Sect. 6, and firstly emphasize the functioning of the variable size DEC scheme.

5.1 DEC Step 1 for Variable Size Length Object

Let $\mathcal{F}_1$ be the first version of a file of size V units. The system distributes the file contents into several *chunks*, each of size Δ units. Within each chunk, $\delta < \Delta$ units of zero pads are allocated at the end while the rest of it are dedicated for the file content. Thus, the V units of the file are spread across

$$M = \left\lceil \frac{V}{\Delta - \delta} \right\rceil \tag{21}$$

chunks $\{C_1, C_2, \ldots, C_M\}$, where $\lceil \cdot \rceil$ denotes the ceiling operator. The zero pads added at the end of every chunk promote sparsity in the difference between two successive versions.

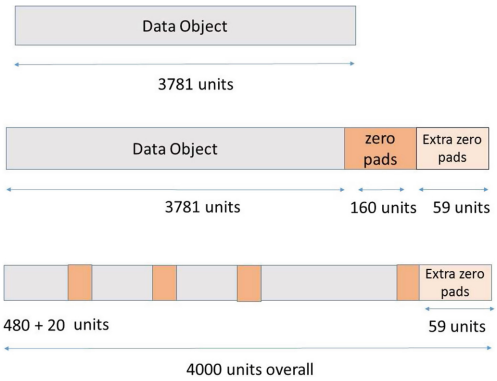

Fig. 14. File structure with different placements of zero pads (ZP) - (i) *ZP-End* where the zero pads are concentrated at the end (middle figure), and (ii) *ZP-Intermediate* where the zero pads are distributed across the file (bottom figure).

Once the file contents are divided into M chunks, they are stored across different servers, using an (n, k) erasure code: the code is applied on a block of k data chunks to output $n(> k)$ chunks which includes the data chunks and $n - k$ encoded chunks that are generated to provide fault tolerance against potential failures. The parameter k is optimized for the architecture with respect to M, which is file dependent:

Case 1: When $M < k$, additional $M - k$ chunks containing zeros are appended to create a block of k chunks. Henceforth, these additional chunks are referred to as *zero chunks*. Then, the k chunks are encoded using an (n, k) erasure code.

Case 2: When $M \geq k$, the M chunks are divided into $G = \lceil \frac{M}{k} \rceil$ groups $\mathcal{G}_1, \mathcal{G}_2, \ldots, \mathcal{G}_G$. The last group $\mathcal{G}_G$ if found short of k chunks is appended with *zero-chunks*. The k chunks in each group are encoded using an (n, k) erasure code.

For the first version $\mathcal{F}_1$, the G groups of chunks together have $\delta M + N\Delta$ units of zero pads, where $1 \leq N < k$, represents the number of zero-chunks added to make $\mathcal{G}_G$ contain k chunks. In addition, the M-th chunk may have extra padding due to the rounding operation in (21). The δM units of zero pads that are distributed across the chunks shield propagation of changes across chunks when an insertion is made in subsequent file versions. This object can now withstand a total of δM units of insertion (anywhere in the file if $\delta M < N\Delta$) by retaining G groups for the second version.

We next discuss the use of zero pads while storing the $(j+1)$-th version $\mathcal{F}_{j+1}$ of the file, $j \geq 1$.

5.2 DEC Step $j + 1$ Under Insertions and Deletions

For the $(j + 1)$-th version, the DEC system is designed to identify the difference in the file content size in every chunk. Then the changes in the file contents are

carefully updated in the chunks, in the increasing order of the indices $1, 2, \ldots, M$, so as to minimize the number of chunks modified due to changes in one chunk. For $1 \leq i \leq M$, if the content of C_i grows in size by at most δ units, then some zero pads are removed to make space for the expansion. This C_i will have fewer zero pads than the first version. On the other hand, if the content of C_i grows in size by more than δ units, then the first Δ units of the file content are written to C_i while the remaining units are shifted to C_{i+1}. The existing content of C_{i+1} is in turn shifted, and hence, it will have fewer zero pads than δ. The propagation of changes in the chunks continue until all the changes in the file are reflected. If the insertion size is large enough, then new chunks (or even new groups) have to be added to the existing chunks (or groups), thus changing the object size of the $(j+1)$-th version.

When file contents are deleted, the zero pads continue to block propagation, this time in the reverse direction. Since deletion results in reduced size of the file contents in chunks, this is equivalent to having additional zero pads (of the same size as that of the deleted patch) in the chunks along with the existing zero pads. After this process, the metadata should reflect the total size of the file contents (potentially less than $\Delta - \delta$) in the modified chunk. Thus, deletion of file contents boosts the capacity of the data structure to shield larger insertions in the next versions.

5.3 Encoding Difference Objects

Note that the differential encoding strategy requires two successive versions to have the same object size to compute the difference. In reverse DEC, once the contents of the $(j+1)$-th version is updated to the chunks, we compute the difference between the chunks of the j-th and the $(j+1)$-th version. Then we declare a difference chunk to be non-zero if it contains at least one non-zero element. Within a group, if the number of non-zero chunks, say γ of them, is smaller than $\frac{k}{2}$ then the difference object is compressed to contain 2γ chunks. We continue this procedure of storing the difference objects until the modified object size is at most kG chunks.

A set of consecutive versions of the file that maintains the same number of groups is referred to as a *batch* of versions, while the number of such versions within the batch is called the depth of the batch. The case when insertions change the group size is addressed next as a source for resetting the differential encoding strategy.

5.4 Criteria to Reset DEC

Criterion 1: Starting from the second version, the process of storing the difference objects continues until G remains constant. When the changes require more than G groups, i.e., the updates require more than kG chunks, the system terminates the current batch, and then stores the object in full by redistributing the file contents into a new set of chunks. To illustrate this, let the j-th version of the file (for some $j > 1$) be distributed across M_j chunks, where $\lceil \frac{M_j}{k} \rceil \leq G$.

Now, let the changes made to the $(j+1)$-th version occupy M_{j+1} chunks where $\lceil \frac{M_j}{k} \rceil > G$. At this juncture, we reorganize the file contents across several chunks with δ units for zero pads (as done for the first version). After re-initialization, this file has $G' = \lceil \frac{M_{j+1}}{k} \rceil$ groups.

Criterion 2: Another criterion to reset is when the number of non-zero chunks is at least $\frac{k}{2}$ within every group. Due to insufficient sparsity in each group, there would be no saving in storage size in this case, and as a result, a new batch has to be started. However, a key difference from criterion 1 is that the contents of the chunks are not reorganized since the group size has not changed.

6 Experiment Results: Performance of Practical DEC

In this section, we present the performance of the practical DEC technique against a wide spectrum of realistic (but synthetically generated) workloads — that include insertions and deletions, which may lead to change in the overall file size, and so on. We also experiment with real world workloads, specifically, we consider multiple versions of Wikipedia documents to drive our experiments with real data.

Its worth reemphasizing at this juncture, that for these experiments, the update model thus doesn't follow Eq. (2), instead it is as per its practical variant which includes zero pads in the file structure. We showcase experiment results with several workloads capturing wide spectrum of realistic loads to demonstrate the efficacy of our scheme. The main objectives are:

1. to determine the right strategy to place the zero pads in order to promote sufficient sparsity in the difference object for different classes of workloads. This objective is achieved using synthetic workloads of insertions and deletions (see Subsects. 6.1 and 6.2).
2. to present the performance of practical DEC with online datasets such as different versions of Wikipedia pages (see Subsect. 6.3).
3. to compare the storage savings of practical DEC against four baselines, namely (i) a naive technique where each version is fully coded and treated as distinct objects, referred to as non-differential scheme, (ii) selective encoding scheme, a system setup which is fundamentally a delta encoding technique where only the modified chunks are erasure coded and then stored, (iii) Rsync, a well known delta encodng technique for file transfer and synchronization across networks, and finally (iv) gz compresssion, which is applied on individual versions to reduce the storage size (see Subsect. 6.4).

Throughout this section, we use the reverse differential method where the order of storing the difference vectors is reversed as $\{\mathbf{z}_2, \mathbf{z}_3, \ldots, \mathbf{z}_L, \mathbf{x}_L\}$. Also, DEC scheme refers not to the primitive form discussed in Sect. 2, but instead it refers to its variant which was discussed in Sect. 5. Unless specified otherwise, we showcase only the best case storage benefits that come with the application of $\frac{k}{2}$-level DEC scheme, wherein the $\frac{k}{2}$ erasure codes are assumed to have identical

storage-overhead of $\kappa = 2$. For the DEC scheme storing two versions, i.e., $L = 2$, the average storage size for the second version is given by

$$\mathbb{E}[\delta(\mathbf{z}_2)] = \kappa \mathbb{E}[\min(2\gamma_j, k)], \tag{22}$$

which is the average size of the data object after erasure coding. When the storage-overhead κ is held constant for all the $\frac{k}{2}$ erasure codes, we note that the quantity

$$\frac{\mathbb{E}[\delta(\mathbf{z}_2)]}{\kappa} = \mathbb{E}[\min(2\gamma_j, k)], \tag{23}$$

which is the average storage size prior to erasure coding, is a sufficient statistic to evaluate the placement of zero pads. Henceforth, we use (23) as the yardstick in our analysis. However, in general, when storage-overheads are different, $\mathbb{E}[\delta(\mathbf{z}_2)]$ in (22) is a relevant metric for the analysis. Notice that unlike the quantities in Subsect. 3.3, the quantity in (23) includes raw data as well as zero pads, and this is attributed to a more realistic model of erasure coded versioning system in Sect. 5, where the zero pads facilitate block encoding of arbitrary sized data objects in addition to shielding the rippling effect from insertions and deletions.

6.1 Comparing Different Placements of Zero Pads

We conduct several experiments to compare the storage savings from the zero pads placements highlighted in Fig. 14. The parameters for the experiment are

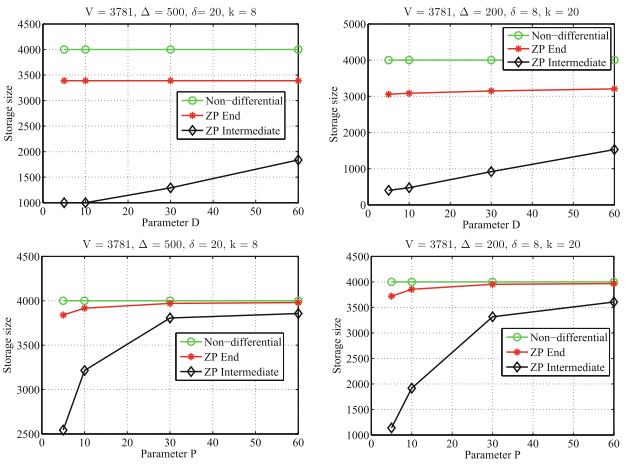

Fig. 15. Comparing different placements of zero pads against insertions: average storage size (as given in (23)) for the 2nd version against workloads comprising random insertions. For the top figures, workloads are bursty insertions whose size is uniformly distributed in the interval $[1, D]$ for $D \in \{5, 10, 30, 60\}$. For the bottom figures, workloads are several single unit insertions whose quantity is distributed uniformly in the interval $[1, P]$, where $P \in \{5, 10, 30, 60\}$.

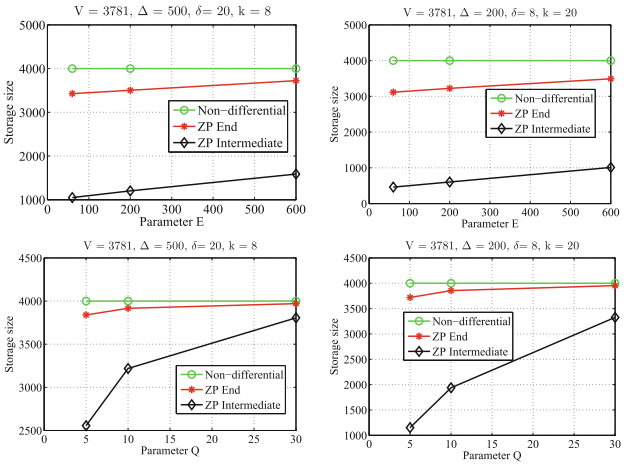

Fig. 16. Comparing different placements of zero pads against deletions: average storage size (as given in (23)) for the second version against workloads comprising random deletions. For the top figures, workloads are single bursty deletions whose size is uniformly distributed in the interval $[1, E]$ for $E \in \{60, 200, 600\}$. For the bottom figures, workloads are several single unit deletions whose quantity is distributed uniformly in the interval $[1, Q]$, where $Q \in \{5, 10, 30\}$.

$V = 3781$, $\Delta = 500, \delta = 20$ and $k = 8$. The two schemes under comparison are *ZP-End* and *ZP-Intermediate* (discussed in Fig. 14), where the zero pads are allocated at the end and at intermediate positions, respectively. Like *ZP-Intermediate* scheme, the *ZP-End* scheme also contains $k = 8$ chunks (each of size Δ), however in this case, 219 zero pads appear at the end in the 8-th chunk. In general, appending zero pads at the end of the data object is a necessity to employ erasure codes of fixed block length. Thus, for the parameters of our experiment, both the *ZP-End* and *ZP-Intermediate* schemes initially have equal number of zero pads (but at different positions), and hence, the comparison is fair.

From our experiments, we compute the average numbers in (23) when two classes of random insertions are made to the first version, namely: (i) single bursty insertion whose size is uniformly distributed in the interval $[1, D]$, for $D = 5, 10, 30, 60$, and (ii) several single unit insertions uniformly distributed across the object, where the number of insertions is uniformly distributed in the interval $[1, P]$, where $P = 5, 10, 30, 60$. We repeat the experiments 1000 times by generating random insertions and then compute the average storage size of the compressed object $\mathbf{z}'_2$ (as given in (23)). In Fig. 15 we plot the average storage size with the *ZP-End* and *ZP-Intermediate* schemes. Similar plots are also presented in Fig. 15 (on the right) with parameters $\Delta = 200, \delta = 20$ and $k = 20$ for the same object. The plots highlight the advantage of distributing the zero pads as it can arrest the propagation of changes through intermediate zero pads. We conduct more experiments for several classes of random deletions

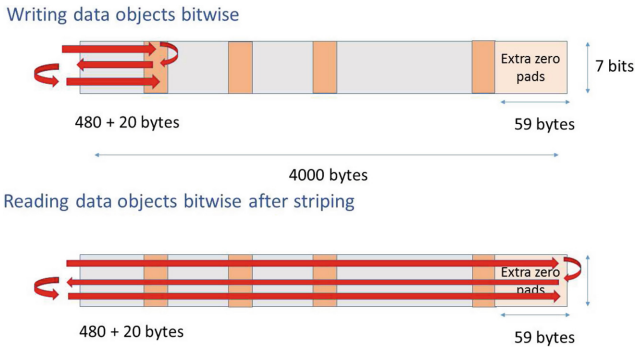

Fig. 17. Bit striping method to generate striped chunks. Top figure depicts bit-level writing of data into the chunks. Bottom figure depicts bit-level reading of data. This technique is suitable for uniformly distributed sparse insertions.

and the results are presented in Fig. 16, which highlight the savings in storage size for the *ZP-Intermediate* scheme.

6.2 Chunks with Bit Striping Strategy

In this section, we analyze the right strategy to synthesize chunks for workloads that involve several single insertions with sufficient spacing. We first explain the motivation for this special case using the following toy example. Consider storing a data object of size $V = 3871$ units using the parameters $\Delta = 500, \delta = 20, k = 8$. Assume that 3 units of insertions are made to the object at the positions $1, 481$ and 961, which translates to modifications of the chunks C_1, C_2 and C_3, respectively. Thus, due to just 3 single unit insertions, three chunks are modified because of which the difference object after compression will be of size 3000 units. Instead, imagine striping every chunk into k partitions at the bit level such that the δ zero pads are equally distributed across the partitions (see the top figure in Fig. 17). Then, create a new set of k chunks as follows: create the t-th chunk for $1 \leq t \leq k$ by concatenating the contents in the t-th partition of all the original chunks (see the bottom figure in Fig. 17). By applying this striping method to the toy example, we see that only one chunk (after striping) is modified, hence, this strategy would need only 1000 units for storage after compression.

For the above example, the insertions are spaced exactly at intra-distance $\Delta - \delta$ units to highlight the benefits, although in practice, the insertions can as well be approximately around that distance to reap the benefits. We conduct experiments by introducing 3 random insertions into the file, where the first position is chosen at random while the second and the third are chosen with intra-distance (with respect to the previous insertion) that is uniformly distributed in the interval $[\Delta - \delta - R, \Delta - \delta + R]$ when $R \in \{40, 80, 120\}$. For this experiment, the average storage size for the second version (i.e., the size of the compressed

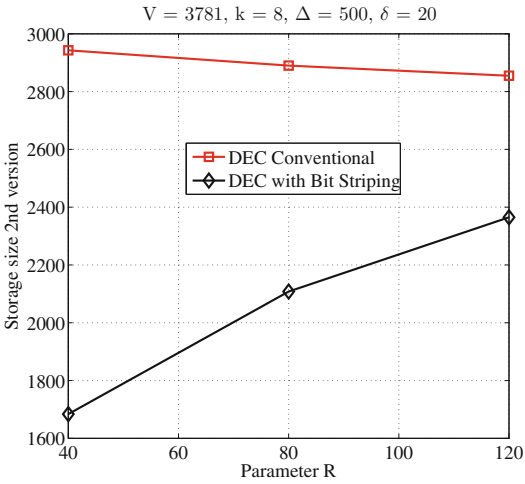

Fig. 18. Comparison of DEC schemes with and without bit striping. Average storage size (given in (23)) for the second version against workload that has 3 single unit insertions with intra-distance uniformly distributed in the interval $[\Delta-\delta-R, \Delta-\delta+R]$, where $R \in \{40, 80, 120\}$. For the experiments, we use $\Delta = 500$ and $\delta = 20$.

object $\mathbf{z}'_2$ given in (23)) is presented in Fig. 18, which shows significant reduction in storage for the striping method when compared to the conventional method. Notice that as R increases, there is higher chance for the neighboring insertions to not fall in the same partition number of different chunks, thus diminishing the gains.

We also test the striping method against two types of workloads, namely, the bursty insertion (with parameter $D \in \{5, 10, 30, 60\}$) and the randomly distributed single insertions with parameter $P \in \{5, 10, 30, 60\}$. For the workloads with single insertions, the spacing between the insertions is uniformly distributed and not necessarily at intra-distance $\Delta-\delta$. In Fig. 19, we present the average storage size for the second version (given in (23)) against such workloads. The plots show significant loss for the striping method against the former workload (as they are not designed for such patterns), whereas the storage savings are approximately close to the conventional method against the latter workload. In summary, if the insertion pattern is known to be distributed a priori, then we advocate the use of the striping method as it provides similar performance as that of the conventional method with a potential to provide reduced storage savings for some special distributed insertions.

6.3 Performance of DEC with Online Datasets

We have conducted experiments based on 5 versions of Wikipedia data on the main article on United States [26], where versions with time stamp "06:21, 15 August 2015" and "00:22, 17th August 2015"are treated as the first

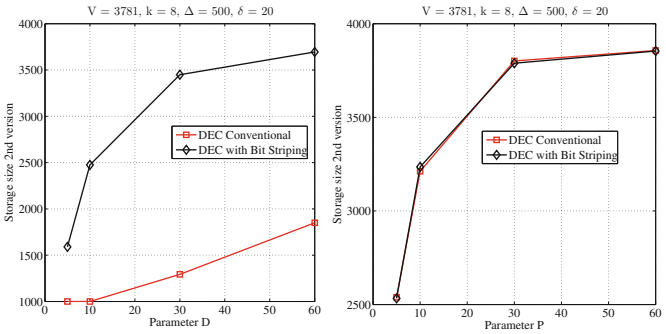

Fig. 19. Comparing DEC schemes with and without bit striping against bursty (the left plot) and randomly distributed single insertions (the right plot) with parameters $D, P \in \{5, 10, 30, 60\}$.

and the fifth versions, respectively. The system parameters for this experiment were

- chunk size (Δ) - 500 bytes
- zero pads in every chunk (δ) - 20 bytes
- number of encoding blocks for erasure coding (k) - 8 chunks

With the above parameters, raw data of 330075 bytes for the first version is expanded to a total of 344000 bytes (by zero pads) and then spread across 86 groups of encoding blocks, where each encoding block contains 8 chunks. For the subsequent versions of the object, changes in different chunks are appropriately identified before storing the difference object as per the DEC scheme. From Version 1 to Version 5, the nature of changes on the chunks are captured in Table 3. For the experiments, we use the reverse differential encoding of Sect. 3 wherein the latest version is encoded in full whereas the preceding versions are stored as difference objects.

Before we proceed to present the storage savings of DEC, the reader may quickly want to know some relevant alternatives for comparison. For cloud storage applications, one straightforward option is to store different versions as

Table 3. Nature of changes at the chunk-level on Wikipedia dataset. The first version has 86 groups of chunks with parameters $\Delta = 500$, $\delta = 20$ and $k = 8$.

Version number	Changes	Affected chunks	Affected groups
1	-	-	-
2	162 bytes removed	5 and 6	1
3	162 bytes added	5 and 6	1
4	102 bytes added	119–124	15 and 16
5	123 bytes added	109–115	14 and 15

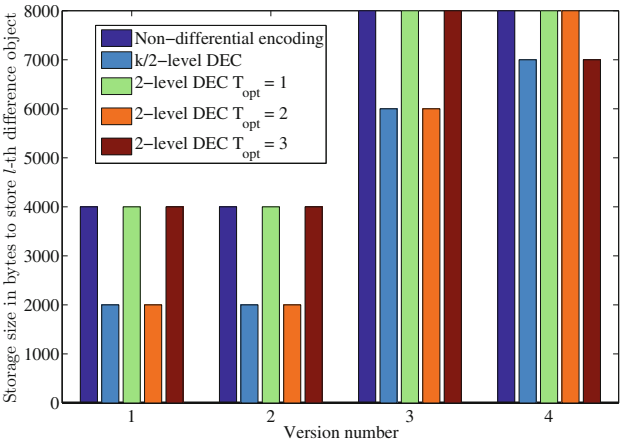

Fig. 20. Performance of DEC on 5 versions of Wikipedia dataset: Storage size (in bytes) needed to store the l-th version, for $1 \leq l \leq 4$ at the end of 5th version, for the following schemes (i) non-differential encoding, (ii) $\frac{k}{2}$-level DEC, (iii) 2-level DEC, with $T_{\text{opt}} = 1$(iv) 2-level DEC with $T_{\text{opt}} = 2$ and (v) 2-level DEC with $T_{\text{opt}} = 3$. Note that the 5th version being the latest is encoded in full, and hence is not presented in the plot.

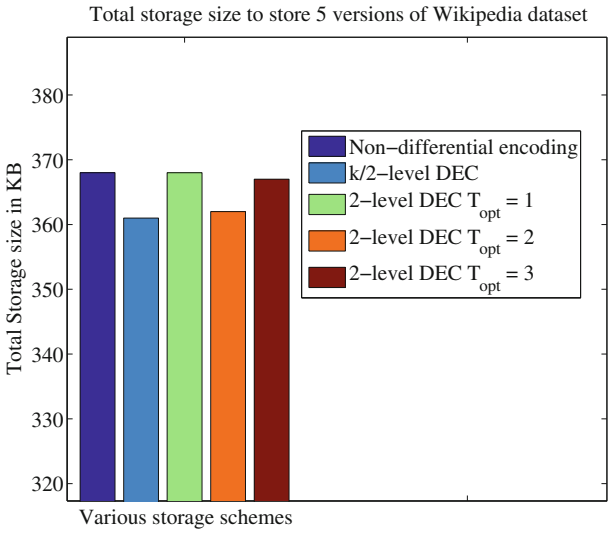

Fig. 21. Performance of DEC on 5 versions of Wikipedia dataset: Total storage size (in KB) needed to store all the 5 versions. The plots indicate that the 2-level DEC scheme with $T_{\text{opt}} = 2$ provides storage savings close to the $\frac{k}{2}$-level DEC.

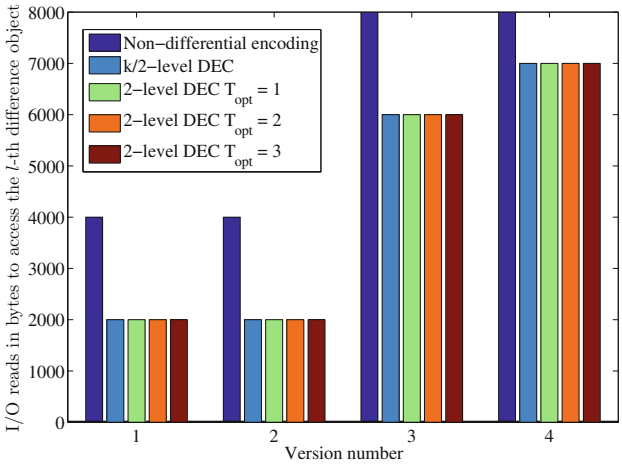

Fig. 22. Performance of DEC on 5 versions of Wikipedia dataset: I/O reads (in bytes) needed to retrieve the l-th version, for $1 \leq l \leq 4$ at the end of 5th version. Although the schemes under comparison provide different storage savings (as shown in Fig. 21), their I/O capabilities are the same due to the use of Cauchy matrices as discussed in Sect. 4.

standalone objects without capitalizing on the *correlation* between successive versions. Since each version is encoded independently, zero pads are needed at the end of the object only to generate the required number of chunks (to be multiple of $k = 8$) for erasure coding. Evidently this method requires around 5 times the size of the first version. Thus, independent encoding of versions along with the insertion of zero pads at the end of the object performs poorly in storage savings, and hence we do not present these numbers. To improve upon this naive scheme, an alternative is to place some zero pads at the end of every encoding group (in the k-th chunk), and then apply non-differential encoding on every modified block. We refer this as the non-differential method in this section.

In Fig. 20, we compare the storage size needed to store the difference objects of the l-th version for $1 \leq l \leq 4$, at the end of 5 versions for the non-differential and the DEC methods. Under the DEC methods, performance of 2-level schemes of Sect. 4 are also presented in Fig. 20, which shows that $T_{\text{opt}} = 2$ provides storage savings close to that of the $\frac{k}{2}$-level DEC scheme. Note that since there are no single-chunk changes, the 2-level DEC scheme with $T_{\text{opt}} = 1$ is as sub-optimal as the non-differential scheme. In Fig. 21, we present the total storage size offered by these schemes to store the 5 versions of Wikipedia dataset. Also, in Fig. 22, we present the I/O performance of the DEC schemes in order to retrieve the difference objects. The figure shows that although the 2-level DEC schemes are not efficient in storage size, their I/O performance is as good as the $\frac{k}{2}$-level DEC. This observation is consistent with the theory that erasure codes from Cauchy matrices (discussed in Subsect. 4.5) help reduce I/O reads when retrieving a sparse data object. Overall, Figs. 20, 21 and 22 confirm that DEC

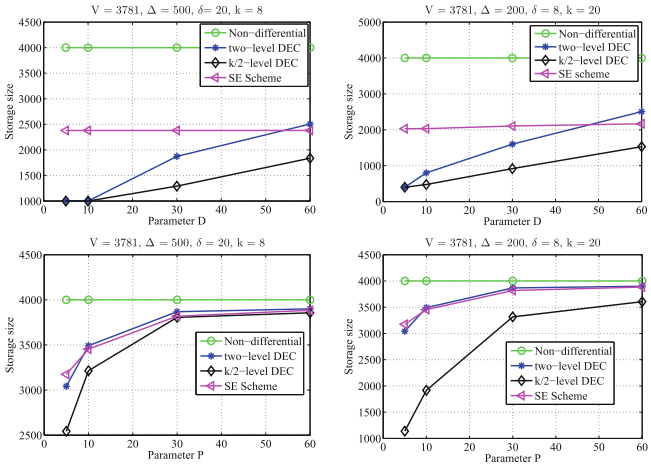

Fig. 23. DEC vs. Selective Encoding with respect to insertions: average storage size for the 2nd version against workloads comprising random insertions. The parameters D and P are as defined for Fig. 15. The $\frac{k}{2}$-level DEC scheme applies an erasure code for each sparsity level, whereas the two-level DEC applies only two erasure codes based on the threshold T_{opt}. The left and the right plots are for bursty and distributed single insertions, respectively.

provides significant reduction in storage size for storing the difference objects without having to look for the modified chunks in every group.

6.4 Comparison with Standard Baselines

Selective Encoding. An important baseline for comparison is a system setup called Selective Encoding (SE), which is a delta encoding technique where only the modified chunks between the successive versions are erasure coded and saved, i.e., visualize the two versions as arrays of numbers, then zero pad the shorter version to make their length equal, and finally store only the changed numbers after component-wise comparison. Observe that the SE scheme is effective if the updated version retains its size, and has few in-place modifications. However, if the object size changes due to insertions or deletions, or when changes propagate at bit level, then dividing the updated version into fixed size chunks need not result in sufficient sparsity in the difference object across versions. For the SE scheme, although there are no preallocated zero pads, they indirectly appear at the end to generate k (or its multiple) number of chunks. We conduct more experiments to compare the storage savings offered by DEC and SE. This time the parameters of the experiment are $V = 3871$, $\Delta = 500, \delta = 20, k = 8$, and the workload includes random insertions with the same parameters as that for Fig. 15. Similar to the preceding experiments, in this section the storage size of the second version includes raw data and zero pads. For the SE based method, zero pads appear at the end to generate $k = 8$ number of chunks from $V = 3871$

units of data. Since, for this experiment, the total number of zero pads is held constant for the two schemes, the comparison is fair. In addition to showcasing the savings of DEC, we present in Fig. 23 the savings of the two-level DEC scheme where only two erasure codes are employed to cater different levels of sparsity. For such a case, the threshold T_{opt} is empirically computed based on the insertion distribution. The plots presented in Fig. 23 highlight the storage savings of both the $\frac{k}{2}$-level DEC and two-level DEC with respect to SE, against bursty insertions (with parameter D). However, for distributed single insertions (with parameter P), only the $\frac{k}{2}$-level DEC outperforms SE, but not the two-level DEC.

Rsync. An advanced version of SE is a storage scheme with concepts from Rsync [19], a delta encoding technique for file transfer and synchronization across networks. The key idea behind Rsync is the rolling checksum computation, using which only the modified/new blocks between successive versions are transferred, thereby reducing the communication bandwidth. With the application of Rsync idea to store versioned data, checksums (or signatures) would have to be computed on every chunk of the j-th version before communicating them to the server containing the $(j+1)$-th version. Subsequently, the server containing latest version rolls over the entire file at fine granularity in search of existing chunks by comparing their checksums, akin to sliding window concept. Finally, the offsets of the found chunks (w.r.t to their position indices in the new file) are returned

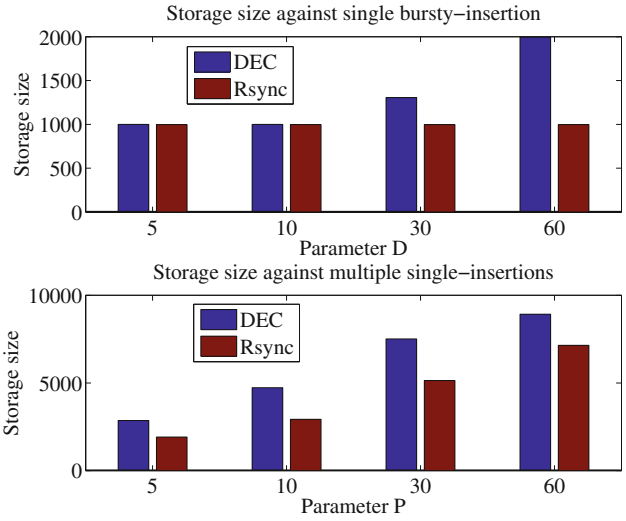

Fig. 24. DEC vs. Rsync with respect to insertions: average storage size for the 2nd version against workloads comprising random insertions. The parameters D and P are as defined for Fig. 15. The plots indicate that Rsync outperfoms DEC in terms of storage savings against both single bursty-insertions and multiple single-insertions.

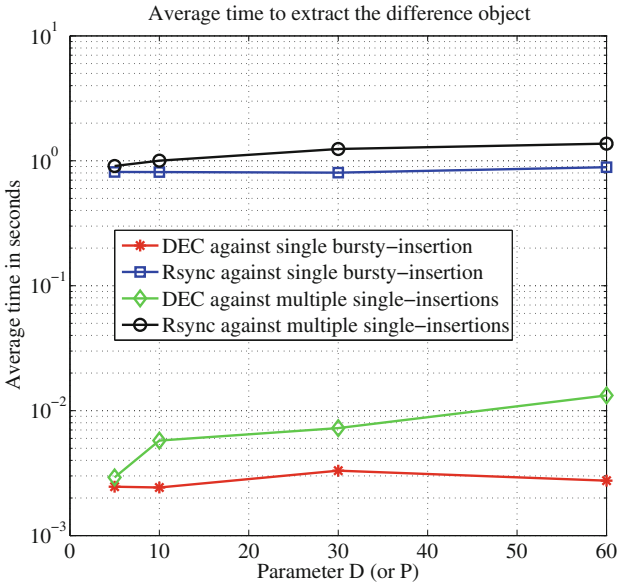

Fig. 25. DEC vs. Rsync with respect to insertions: average time (in seconds) to extract the difference object from the second version in experiments against workloads comprising random insertions. The Rsync scheme consumes substantial time to extract the difference object as it has to apply the sliding-window algorithm at the byte-level in order to look for duplicated chunks in the second version. However, the DEC scheme is computationally efficient due to one-time subtraction of the two versions of data object.

along with the new file contents and corresponding offsets. We note that the rolling checksum computation, which works at unit-level granularity across the file, can be thought as a replacement to the low-complexity subtraction operation in DEC. However, the advantage of reduced computational complexity of DEC comes at the cost of additional storage size for zero pads. Also, since insertions and deletions appear at arbitrary positions in the file, metadata for Rsync should harbor offsets (a.k.a positions of new contents in the file) at unit-level granularity. However, such information is stored at the chunk-level granularity in DEC, thus making it relatively simple in terms of metadata management.

We have conducted experiments to compare the performance of Rsync and DEC. The system parameters for the experiments were $V = 8740$ units, $\Delta = 500$, $\delta = 20$ and $k = 8$. The experiments were conducted against workloads comprising random insertions characterized by single bursty-insertions with parameter $D \in \{5, 10, 30, 60\}$, and multiple single-insertions with parameter $P \in \{5, 10, 30, 60\}$. The average storage size of the difference objects for the two methods are presented in Fig. 24, which shows that Rsync outperforms DEC. This observation can be attributed to the fact that the rolling checksum algorithm of Rsync is powerful enough to search for duplicated chunks in the

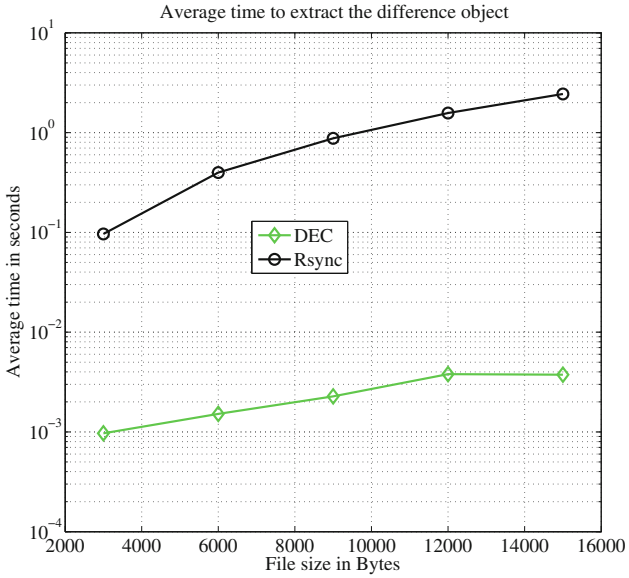

Fig. 26. DEC vs. Rsync with respect to insertions: average time (in seconds) to extract the difference object from the second version in experiments against insertion workloads of different file sizes.

second version, except for those where the insertions were made. Specifically, the initial file of size $V = 8740$ units is first broken into 18 chunks each of size 500 units. Subsequently, each one of the 18 chunks is searched for duplication in the second version by a sliding window comparison that involves 18×8240 chunk-level subtractions in the worst-case. On the other hand, the DEC scheme involves only 24 chunk-level subtractions to obtain the difference object. Thus, the Rsync scheme provides reduced storage size than DEC by trading-off computational complexity. In general, if the file size is V units and the number of chunks is C, then the total number of chunk-level subtractions for Rysnc is of the order $O((V + I)C)$, with I being the insertion size. whereas the corresponding number in DEC is O(C), assuming I is less than that the total number of zero pads. To capture the difference in the computation time between the two schemes, we measure the average time to extract the difference objects against random insertions. The software routines for extracting difference objects (for both Rsync and DEC) were implemented on 64 bit Intel(R) Core(TM) processor @2.13 GHz. The measured average time duration are presented in Figs. 25 and 26, which show a significant difference in the processing time of the two schemes. Overall, we summarize the differences between Rsync and DEC in Table 4. Other than the storage size and computational complexity features, we have also listed erasure coding management as a distinguishing feature. Note that DEC has provision for using just two erasure codes to cater to different sparsity levels, and so, erasure coding management is easy. On the other hand, since Rsync extracts

Table 4. Summary of comparison between Rsync and DEC

Feature	DEC	Rsync
Storage size	Low	Lower than DEC
Computational complexity	Low	High
Erasure coding management	Easy	Complex

only the modified blocks (that are of arbitrary size), it explicitly requires an erasure code for every sparsity level especially when the symbol-size for erasure coding is fixed.

Compression Techniques. One more baseline for comparison is from the set of standard file compression algorithms that are employed to store different versions of a data object. Typically, a file compression algorithm exploits redundancy within the file to generate a compressed file of considerably smaller size. In contrast to such compression schemes, DEC scheme exploits redundancy across versions instead of redundancy within a single version. In order to compare the two schemes, we use the Wikipedia dataset of Subsect. 6.3 to compute the storage savings of the two schemes. We use $\{V_1, V_2, \ldots, V_5\}$ to denote those 5 raw versions. Then, we use the *gz* compression available online at [25] to first generate the compressed counterparts of the 5 versions of Wikipedia pages, namely $\{W_1, W_2, \ldots, W_5\}$. The file sizes of the 5 versions before and after *gz* compression are given in the 2nd and 3rd columns of Table 5, respectively, which show that compressed versions are close to 50% of the original size. Subsequently, we compute the sparsity across subsequent versions of the compressed objects, i.e., sparsity of $\{W_1 - W_2, W_2 - W_3, W_3 - W_4, W_4 - W_5\}$, and then compare those

Table 5. Sparsity across successive versions of the Wikipedia dataset in Subsect. 6.3. Compressed versions of the raw data are obtained using [25]. Although the changes on raw data are fewer, compression of individual versions does not promote sparsity across compressed versions. Observe that the % of non-zeros is higher across subsequent compressed versions. Here, low % of non-zero entries implies high sparsity.

Version number	Size of raw data in Bytes	Size of compressed data in Bytes	% of non-zeros across successive versions of raw data of (high sparsity)	% of non-zeros across successive versions compressed data (low sparsity)
1	330593	157068	-	-
2	330429	157060	0.30%	98.41%
3	330593	157068	0.302%	98.41%
4	330701	157096	0.9%	98.21%
5	330824	157148	1.05%	81.45%

values with that obtained from the raw Wikipedia data. For the compressed objects, sparsity values are computed by suitably zero padding the shorter of the two versions. For instance, W_2 is appended with 8 bytes of zero pads to compute the sparsity w.r.t W_1. Although the gz compression algorithm reduces the size of each version, it mixes the contents within each version in such a way that subsequent versions differ at majority of positions despite few in-place alterations on raw data. Thus, % of zeros across subsequent compressed versions are expected to be high, and therefore, applying DEC on the compressed objects may not bring any more reductions in the storage size. On the other hand, we have already shown that DEC enables to store the raw versions at lower storage-size due to high sparsity across raw versions. In Table 5, we present the % of non-zero entries in the difference objects of both raw and compressed data, and the numbers indicate that DEC schemes are effective on raw data than on compressed versions. Thus, the total storage size offered by DEC is substantially smaller than that by individual compression.

7 Concluding Remarks

This paper proposes differential erasure coding techniques for improving storage efficiency and I/O reads while archiving multiple versions of data. Our evaluations demonstrate tremendous savings in storage. Moreover, in comparison to a system storing every version individually, the optimized reverse DEC retains the same I/O performance for reading the latest version (which is most typical), while reducing significantly the I/O overheads when all versions are accessed, in lieu of minor deterioration for fetching specific older versions (an infrequent event). Future works aim at integrating the proposed framework to full-fledged version management systems.

Acknowledgements. This work was supported by the MoE Tier-2 grant "eCode: erasure codes for data center environments" (MOE2013-T2-1-068).

References

1. Huang, C., Simitci, H., Xu, Y., Ogus, A., Calder, B., Gopalan, P., Li, J., Yekhanin, S.: Erasure coding in windows azure storage. In: The Proceedings of the USENIX Annual Technical Conference (ATC) (2012)
2. Thusoo, A., Shao, Z., Anthony, S., Borthakur, D., Jain, N., Sen Sarma, J., Murthy, R., Liu, H.: Data warehousing and analytics infrastructure at Facebook. In: The Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data (2010)
3. Ford, D., Labelle, F., Popovici, F.I., Stokely, M., Truong, V.A., Barroso, L., Grimes, C., Quinlan, S.: Availability in globally distributed storage systems. In: The 9th USENIX Conference on Operating Systems Designand Implementation (OSDI) (2010)
4. Dimakis, A.G., Ramchandran, K., Wu, Y., Suh, C.: A survey on network codes for distributed storage. Proc. IEEE **99**, 476–489 (2011)

5. Oggier, F., Datta, A.: Coding techniques for repairability in networked distributed storage systems. In: Foundations and Trends in Communications and Information Theory, vol. 9, no. 4, pp. 383–466. Now Publishers, June 2013
6. http://subversion.apache.org/
7. http://www.ibm.com/developerworks/tivoli/library/t-snaptsm1/index.html
8. Borthakur, D.: HDFS and Erasure Codes (HDFS-RAID), August 2009. http://hadoopblog.blogspot.com/2009/08/hdfs-and-erasure-codes-hdfs-raid.html
9. The Coding for Distributed Storage wiki. http://storagewiki.ece.utexas.edu/
10. Wang, Z., Cadambe, V.: Multi-version Coding for Distributed Storage. In: Proceedings of IEEE ISIT 2014, Honalulu, USA (2014)
11. Rouayheb, S., Goparaju, S., Kiah, H., Milenkovic, O.: Synchronising edits in distributed storage networks. In: The Proceedings of the IEEE International Symposium on Information Theory, Hong Kong (2015)
12. Rawat, A., Vishwanath, S., Bhowmick, A., Soljanin, E.: Update efficient codes for distributed storage. In: IEEE International Symposium on Information Theory (2011)
13. Han, Y., Pai, H.-T., Zheng, R., Varshney, P.K.: Update-efficient regenerating codes with minimum per-node storage. In: Proceedings of IEEE International Symposium on Information Theory (ISIT 2013), Istanbul (2013)
14. Mazumdar, A., Wornell, G.W., Chandar, V.: Update efficient codes for error correction. In: The Proceedings of IEEE IEEE International Symposium on Information Theory, Cambridge, MA, pp. 1558–1562, July 2012
15. Esmaili, K.S., Chiniah, A., Datta, A.: Efficient updates in cross-object erasure-coded storage systems. In: IEEE International Conference on Big Data, Silicon Valley, CA, October 2013
16. Tarasov, V., Mudrankit, A., Buik, W., Shilane, P., Kuenning, G., Zadok, E.: Generating realistic datasets for deduplication analysis. In: The Proceedings of the 2012 USENIX Conference on Annual Technical Conference (2012)
17. Harshan, J., Oggier, F., Datta, A.: Sparsity exploiting erasure coding for resilient storage and efficient I, O access in delta based versioning systems. In: The Proceedings of IEEE ICDCS, Columbus, Ohio, USA (2015)
18. Harshan, J., Oggier, F., Datta, A.: Sparsity exploiting erasure coding for distributed storage of versioned data. Computing **98**, 1305–1329 (2016). Springer
19. http://rsync.samba.org/
20. Donoho, D.L.: Compressed sensing. IEEE Trans. Inf. Theor. **52**(4), 1289–1306 (2006)
21. Zhang, F., Pfister, H.D.: Compressed sensing and linear codes over real numbers. In: Information Theory and Applications Workshop (ITA) (2008)
22. McWilliams, F.J., Sloane, N.J.A.: The Theory of Error Correcting Codes. North Holland, Amsterdam (1977)
23. Lacan, J., Fimes, J.: A construction of matrices with no singular square submatrices. In: The Proceedings of International Conference on Finite Fields and Applications, pp. 145–147 (2003)
24. Harshan, J., Datta, A., Oggier, F.: DiVers: an erasure code based storage architecture for versioning exploiting sparsity. Future Gener. Comput. Syst. **59**, 47–62 (2016). Elsevier
25. http://www.txtwizard.net/compression
26. https://en.wikipedia.org/wiki/United_States

http://www.springer.com/978-3-662-54053-4

Transactions on Large-Scale Data- and Knowledge-Centered Systems XXX
Special Issue on Cloud Computing
Hameurlain, A.; Küng, J.; Wagner, R.; Schewe, K.-D.; Bosa, K. (Eds.)
2016, IX, 133 p. 62 illus., Softcover
ISBN: 978-3-662-54053-4