Alexandria: A Proof-of-concept Implementation and Evaluation of Generalised Data Deduplication

Lars Nielsen[1], Rasmus Vestergaard[1], Niloofar Yazdani[1], Prasad Talasila[1], Daniel E. Lucani[1], Márton Sipos[2]
[1]Department of Engineering, DIGIT, Aarhus University, Aarhus, Denmark
[2]Department of Automation and Applied Informatics, Budapest University of Technology and Economics, Budapest, Hungary
{lani, rv, n.yazdani, prasad.talasila, daniel.lucani}@eng.au.dk, siposm@aut.bme.hu

***Abstract*—The amount of data generated worldwide is expected to grow from 33 to 175 ZB by 2025 [1] in part driven by the growth of Internet of Things (IoT) and cyber-physical systems (CPS). To cope with this enormous amount of data, new edge (and cloud) storage techniques must be developed. Generalised Data Deduplication (GDD) is a new paradigm for reducing the cost of storage by systematically identifying near identical data chunks, storing their common component once, and a compact representation of the deviation to the original chunk for each chunk. This paper presents a system architecture for GDD and a proof-of-concept implementation. We evaluated the compression gain of Generalised Data Deduplication using three data sets of varying size and content and compared to the performance of the EXT4 and ZFS file systems, where the latter employs classic deduplication. We show that Generalised Data Deduplication provide up to 16.75% compression gain compared to both EXT4 and ZFS with data sets with less than 5 GB of data.**

***Index Terms*—deduplication, edge computing, edge storage**

I. Introduction

Due in part to the Internet of Things (IoT) and the growth of Cyber Physical Systems (CPS), the amount of data generated globally has been drastically increasing. It is estimated that the amount of data worldwide will grow from 33 ZB in 2018 to 175 ZB in 2025 [1]. For data storage systems to accommodate this enormous amount of data, we must not only consider growing current infrastructures, but also develop novel techniques for lossless data compression that can operate in Cloud and Edge computing solutions. This is particularly important for Edge computing as the amount of devices at the edge and their storage capacity is significantly more limited than for Cloud systems. Beyond compression of individual files, data deduplication [2]–[4] constitutes a state-of-the-art technique that addresses some of these challenges. Data deduplication works by recognising reoccurring data patterns in large chunks of data and ensuring that a pattern is stored only once. Deduplication allows the system to access data without the need for decompressing all data in the system, as is the case in most compression algorithms. In general, classic data deduplication works by dividing data, e.g. files, into multiple data chunks and identifying if there are duplicates amongst the chunks already stored in the system ($\mathcal{C}$). Each data chunk c is stored only once. Comparing raw c's bit by bit is an expensive procedure that would grow in complexity as more data is stored in the system. To reduce the cost of chunk comparison, hash functions ($H(\cdot)$) for each chunk c are computed [5]. This means that for every c a unique (with high probability) hash key ($c_{key} = H(c)$) is generated and kept in a registry ($\mathcal{C}_{keys}$). If $c_{key} \notin \mathcal{C}_{keys}$, then we store c, i.e., we add c to the set $\mathcal{C}$, and add c_{key} to $\mathcal{C}_{keys}$. This approach requires the system to keep track of the order of chunks in a file in order to be able to reconstruct the original file. To ensure fast data deduplication storage systems often keep $\mathcal{C}_{keys}$ in memory. Typically, this requires a significant use of memory, e.g., a rule of thumb for ZFS is to have 5 GB of RAM per 1 TB of data stored on disk to support deduplication [6].

This work was partially financed by the SCALE-IoT Project (Grant No. 7026-00042B) granted by the Independent Research Fund Denmark, by the Aarhus Universitets Forskningsfond (AUFF) Starting Grant Project AUFF-2017-FLS-7-1, and Aarhus University's DIGIT Centre.

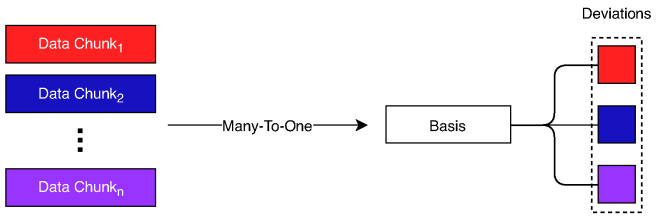

Fig. 1. Mapping data chunks to a basis and deviations

Though classic deduplication reduces the amount of data stored, it has a fundamental limitation: if two chunks deviate by even 1 bit, they will be treated as two different chunks. As a solution to this problem, Generalised Data Deduplication (GDD) was recently developed [7] to systematically map a large amount of similar data chunks to a common basis. The goal of GDD is to increase the compression gain of deduplication, to decrease the size of $\mathcal{C}_{keys}$, and to provide gains with fewer data chunks stored in the system compared to classical deduplication. All these features are critical when storing data at the Edge of the network to deal with resource limited devices. In this paper, we present a system architecture for GDD and a proof-concept implementation, which we evaluate using different configurations and real data sets. We compare GDD against ZFS [8] and show that there is compression gain of 16.75 % for small amounts of data (< 5 GB). Larger gains are expected as the data stored in the system grows [7].

II. Generalised Data Deduplication

As stated earlier the goal of GDD is to increase the compression gain of classical deduplication. To achieve this, GDD employs a *Many-To-Mapping* where multiple chunks c's each of size n bits are mapped to a common basis (b) of size k and a deviation (d) of size p, as illustrated in Figure 1. Deduplication is then conducted over the set of registered bases ($\mathcal{B}$) using the same method as classical deduplication,

by having a registry of hash keys for the bases ($\mathcal{B}_{keys}$) and identify if $H(b)$ is already registered. A key difference is that we need to store the order of bases and their respective deviations for each file. Thus, a file will be a collection of hash of basis and deviation pairs ($(H(b), d)$). We define q as the size in bits for $H(\cdot)$.

To generate the basis and deviations, GDD uses a deterministic transformation function ($T(c)$) and when data is accessed an inverse of the transformation function ($T^{-1}((b, d))$) is needed to reconstruct c. Without loss of generality, we propose the usage of an Error-Correction Code (ECC) in an *unconventional* way for this process, by using the ECC to *decode* c to identify its b and d. To reconstruct c, we then *encode* (b, d) using the ECC. Typically, an ECC would correct bit errors when we send our data via a transmission channel (See Figure 2). Thus, we send not only our data, e.g., `1010`, but also some redundancy, e.g., `110`, resulting in a transmitted sequence, e.g., `1010110`. If bit number 3 is flipped in Figure 2, then the ECC would be able to recover our original data from the data with errors. In coding theory terms, we can calculate the syndrome (s) representing that an error happened at the third bit. In contrast, GDD assumes that a chunk of data, e.g., `1010110`, is first converted (decoded) into a base, e.g., `1010`, and a deviation, e.g, the syndrome s, as in Figure 3. In order to reconstruct, we encode the base and then apply the error indicated by s.

For GDD to provide a higher storage gain that classic deduplication, the combined size of $\mathcal{B}$, $\mathcal{B}_{record}$, and the size of the registry of files as a concatenation of $(H(b), d)$ pairs must be smaller than the combined size of $|\mathcal{C}|$, $|\mathcal{C}_{record}|$, and the size of the registry of files as a concatenation of $H(c)$'s. It is possible to achieve a storage gain when several chunks have the same basis since that basis needs to be stored only at the first occurrence. For subsequent occurrences, a reference to the basis suffices. For each new chunk c_i with basis b_i and deviation d_i, the storage cost will be

$$S(c_i) = I\{b_i \notin \mathcal{B}\}\,(k+q) + q + p \quad [\text{bits}], \tag{1}$$

where $I\{\cdot\}$ is the indicator function defined as $I\{b_i \notin \mathcal{B}\} = 1$ if $b_i \notin \mathcal{B}$ and zero otherwise.

Example: If we assume that bases are drawn independently and uniformly at random, then we can analyze the probability of b_j, the basis of chunk c_j, not being among the previous $j-1$ stored bases as

$$\begin{aligned}\mathbb{P}[b_j \notin \mathcal{B}] &= \mathbb{P}[b_1 \neq b_j \wedge b_2 \neq b_j \wedge \cdots \wedge b_{j-1} \neq b_j] \\ &= \prod_{i=1}^{j-1} \mathbb{P}[b_i \neq b_j] = \prod_{i=1}^{j-1} \frac{|\mathcal{X}|-1}{|\mathcal{X}|} = \left(1 - |\mathcal{X}|^{-1}\right)^{j-1}\end{aligned}$$

where $\mathcal{X}$ is the set of all bases required to represent the data in the system. In the worst case, $\mathcal{X}$ can include all potential bases of length k bits, i.e., $|\mathcal{X}| = 2^k$. This result is valid for classical deduplication if we consider the case where the chunks and the bases are equal (no deviation) and $\mathcal{X}$ would represent the set of all possible values generated by the data source. In the worst case this set would be of size 2^n. This probability converges to 0 as the number of bases grows, so as more data goes into the system it becomes more likely to already have a match. If the set $\mathcal{X}$ is large, then this convergence happens slowly and a larger amount of data would be required. Given that the set of possible chunks is larger or equal than the set of possible bases, $\mathcal{X}$, GDD converges faster than classical deduplication.

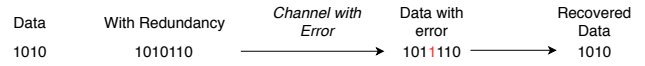

Fig. 2. Typical use of ECC to recover from errors in a transmission channel

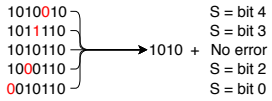

Fig. 3. Use of ECC as a transformation function from each data chunk to a (common) basis and deviation

When there is a basis match, the new chunk costs only $p+q$ bits to store in the system in GDD for the file registry. If there is no match, then the system must store $p+2q+k$ bits, which accounts for the $p+q$ bits for the file registry and $k+q$ bits for storing the basis and the respective hash to it. To have a potential for high gains, $p+q \ll n$.

Thus, GDD's compression gain after storing a total of N chunks that can be represented using $K = |\mathcal{B}|$ bases is

$$G = \frac{Nn}{K(k+q) + N(p+q)}. \tag{2}$$

III. Systems Architecture

In this section we introduce Alexandria, a system architecture for GDD. We present the different components and their role in the architecture and how it relates to the GDD process. Although [7] focused on the usage of Hamming Code [9] as ECC for the transformation, our architecture will support various ECC codes and allow adaptive selection of them based on the data ingested to the system. Figure 4 outlines the system components and their connection to the GDD process. Connections between components show the inputs and outputs to/from each component. Full lines shows connections active when storing data, while dotted lines show active connections when loading/accessing data. As Figure 4 shows, the architecture is divided into three component groups.

Data Transformation Components: provide $T(c)$ and $T^{-1}(b, d)$, but also manage the different transformation functions provided and choose which $T(c)$ is most suited for the data. In Figure 4, we show a collection of *Transformation Function* components, where each represents the implementation of a given $T(c)$ and its inverse. When data is stored, the transformation function is chosen based on the *transformation configuration* provided. The system then divides the data into chunks and applies $T(c)$ to produce the base and deviation pairs for each provided chunk. The transformation configuration depends on the $T(c)$ used, e.g., for Hamming Codes the configuration would contain the message length. When storing data, the component will return the base and deviation pairs. When loading data from the system, the component will again take the transformation configuration as an input along with

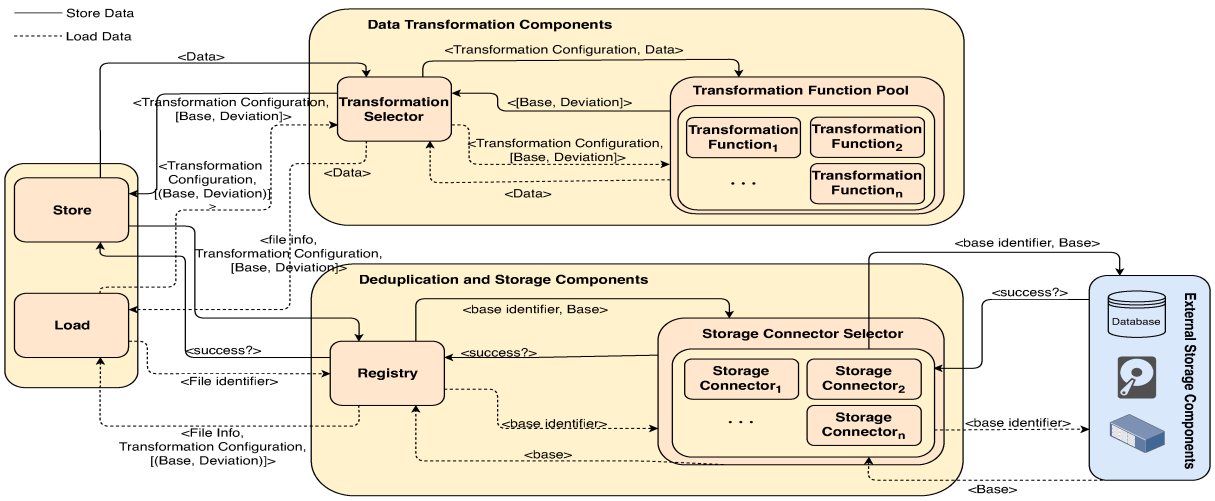

Fig. 4. Alexandria: System Architecture support for Generalised Data Deduplication

the base and deviation pairs, and it will reconstruct the original data and return it.

The *Transformation Pool* serves as a collection and manager of the active/instantiated transformation functions, as shown in Figure 4. The pool will select the correct transformation function and return the response of the transformation function, based on the provided configuration. This is true for both storing and loading data. If the pool does not have an instance of the correct transformation function available, it will queue the task until the an instance is ready. The final transformation component is the *Transformation Selector*. Its taks is to identify what transformation function and configuration best suits the data when data is being stored. It will then parse this information on to the transformation pool and return the response of the pool. The selection algorithm can vary based on the available transformation functions, data size, data type, overall workload to the storage system, among other parameters. An in-depth study of the selection algorithms is left for future work. When storing data, we let the selector serve as a pass through component to maintain the sequence of component calls in the system. When data is being stored, the transformation configuration is returned by the selector. Thus, the configuration can be stored elsewhere in the system.

Deduplication and Storage Components: serve two main purposes, 1) identifying data duplicates, and 2) persistently store data and enable access to it. The *Storage Connector* components in Figure 4 are responsible for providing a uniform API to manage and access external storage systems such as disks, databases, or servers. Additionally, these connectors provide support for basic health checks for each storage system, e.g., is it accessible, how much storage is used. The storage connectors will typically store the bases, i.e., a connector takes a base b with identifier b_{key} and stores it in a given storage system. The connector will the return a success message identifying if the storage procedure was successful or not. When data is being loaded in the system, the storage connector takes b_{key} and returns the corresponding b.

The *Storage Selector* is responsible for maintaining a pool of storage connectors, queue tasks for data storing and loading, and as a mechanism for selecting which storage component will be used to store b. As with the Transformation selector, the choice of storage connectors can vary, e.g., based on the amount of available storage, response time from a connector/system. When data has been stored, the storage selector will return an acknowledge message from the connector and its identifier. When data is being loaded, the selector will take b_{key} and the identifier of the specific storage connector holding the base and request b from the specified connector. If successful, it will return b. Figure 4 illustrates the case of a storage selector and storage connector handling one basis at a time. However, we recommend that each call is able to handle batches of bases when storing and loading data.

Finally, the *Registry* handles deduplication, registration of bases, file information, base and deviation pairs order of the ingested data, and transformation configuration. When it receives the information for storing, the registry first registers the file with some identifier, e.g., filename, unique identifier, and store the file information along side this identifier. For each basis b a b_{key} is generated. The registry component manages a persistent version of $\mathcal{B}_{keys}$. If $b_{key} \notin \mathcal{B}_{keys}$, then b will be stored using the storage selector. If the storage selector returns a success, then b_{key} is added to $\mathcal{B}_{keys}$. We will extend $\mathcal{B}_{\|\rceil\dagger f}$ to not only contain the hash keys, but also the storage connector identifier, such that we alter can return the basis via the storage selector. Next, the basis and deviation pairs's orde in the data/file will be recorded and stored persistently. We propose storing it using a record of tuples with the following format (file identifier, i, (b_{key}, d)), where i is an index identifying the position of (b, d) in the file. The registry will return a success message identifying if the system succeed in storing file. When data is loaded from the system, the registry will take the file identifier as an input to identify the set of bases to be retrieved through the storage selector. When the bases are retrieved, the registry will organise them and associate them with their corresponding deviation. Finally, the registry will return the file information, transformation configuration and the bases and deviation pairs, which will be sent to the transformation components for final recovery of the file.

Public API: provides interface components for storing and loading data from the system, and orchestrates what information the two previous component groups receive. When a file is ingested into the system, the API will first provide the file's data contents to the transformation components. When the transformation configuration and bases and deviations pairs are returned, the API will provide these along side the file information to the deduplication and storage components. When they are done, the API component sends an acknowledgement message to the API user. When loading data, the API first sends the requested file identifier to the deduplication and storage components and waits for them to return the file information, transformation configuration, and bases and deviation pairs. The configuration and the pairs will then be provided to the transformation components. These will reconstruct the data and return it to the API component. Finally, the original file is reconstructed using the data and the file information and returned to the consumer of the API.

Note that we do not prescribe an exact API in Figure 4 for the communication between components. This can be decided during implementation depending on the scope and requirements of the syste. This allows for an architecture that can be implemented using local invocation, invocation over networks, or a combination of the two. Thus, the architecture can be used in both a centralised and distributed manner.

IV. Proof-of-concept Implementation

We implement a highly configurable proof-of-concept in C++ based on the architecture presented and portable to UNIX based/liked systems. We focus on the usage of Hamming Code as our transformation function and a single Edge server as our external storage system.

Configuration of Individual Components: we have set the requirement that components must be configurable at initialisation time using JSON [10], as JSON is easy to adapt, provides a human-readable format, and can be generated at run-time or read from files on disk.

Transformation Components: we have implemented a Hamming Code which is our $T(c)$, and it takes the *message length* as parameter when constructed, and provides functions for *encoding* and *decoding* data. For the proof-of-concept, we provide the message length at application launch to lock the configuration, such that we can evaluate the performance of a specific configuration for the transformation function. With one transformation function is the transformation- selector and function pool irrelevant components, and we have chosen not to implement them. However, we provide a wrapper module which enables us to add more codes in the future, by parsing a JSON object to the module containing a function identifier and its configuration we can generate an instance of the function.

The encoding and decoding speed for the Hamming transformations used is provided in Figure 5 for different chunk sizes associated to the m parity elements of Hamming, where the chunk size is given by 2^m bits. We have used a Core i7 3770 at 4.08 GHz for these tests and focus on a single thread (and CPU) for execution. Figure 5 shows that the speed for low chunk sizes essentially doubles with doubling of the data, which is related to the fact that the chunks are small and

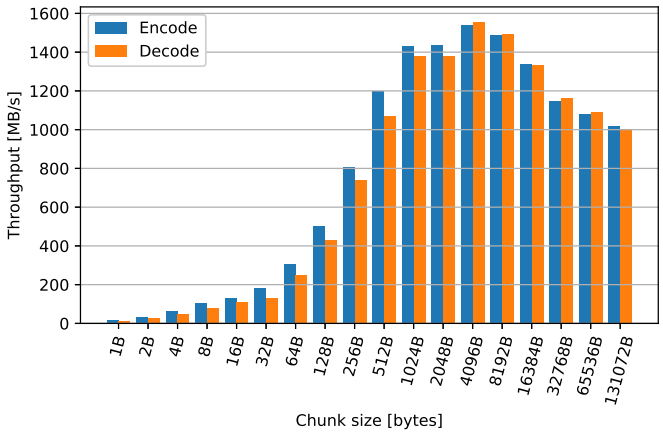

Fig. 5. Processing speed of encode and decode for our implementation of the Transformation function (Hamming Code)

thee is a cost for calling the used methods. The performance peaks at around chunk sizes of 4096 bytes and is reduced somewhat for larger chunks. This degradation is related to the increased number of operations for larger chunks, but show already encoding/decoding speeds over 1000 MB/s (i.e., 8 Gbps) for a wide range of chunk sizes, namely, from 512 to 131072 bytes. These results are not yet optimized for speed and will benefit from further hardware accelleration available in standard CPUs, e.g., Single Instruction Multiple Data, and multithreading options. This acceleration will be the focus of future work.

Storage and Deduplication Components: we have implemented a single storage connector which serves to wrap a directory on disk. It is implemented using the C++ file system library [11] and can be deployed on any file system supported by the library. The registry is a combination of a Postgres [12] database and C++ code. We use the database as the persistent part of the registry to maintain the $\mathcal{B}_{keys}$, the order of basis and deviation pairs in files (this includes storing the deviation in the registry), and the coding configuration. The full database diagram for the registry can be seen in Figure 6. To generate the hash key for the bases, we use the SHA-1 implementation provided with Crypto++ [13], and we use the `UNIQUE` database constraint [14] to ensure that there are no replicated keys in the registry. As we only have one storage connector, we have chosen not to implement a storage selector.

Public API: we have implemented a simple Public API such that it takes a folder as input, and every file in that folder will be stored in the system. To load a file from the system, all that is need to be provided is the name of the file as an identifier.

V. Measuring Storage Space

Before presenting the evaluation of the proof-of-concept, we first address how we compute the storage space used for a standard file system without deduplication (EXT4), a file system with deduplication (ZFS), and Alexandria.

EXT4: We used the standard `du -hb` command to determine the size of the folder(s) containing the original data.

ZFS: we have set up a partition running ZFS to test performance and inserted the same data as for Alexandria and

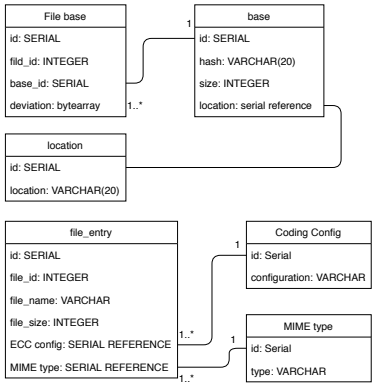

Fig. 6. The tables needed to support a registry for GDD and their references to each other

Data type	Size in bytes
`serial`	4
`integer`	4
`varchar`	1 per character
`bytearray`	1 byte + length of binary string

TABLE I
SUMMARY OF POSTGRESSQL DATA TYPE BYTE SIZES [15]–[17]

in the same order. We used the standard `zpool list` and looked at the `DEDUP` column for the deduplication ratio.

Alexandria: As stated earlier, we store each basis in an individual file. The total storage consumption for bases is then $|\mathcal{B}|(k+m_f)$, where m_f accounts the basis file's meta data size and k is the size of the basis. Additionally, we have to factor in the size of the registry. Figure 6 shows the data types we use and Table I shows the current choice of size of each data type in bytes. We compute the size of a single entry in bytes considering the different elements of our registry, as described in Table II. From this, we can compute the total registry size in bytes as $|\mathcal{B}| \cdot b + \beta \cdot (l + cc + m_t) + \gamma \cdot f_b + \delta \cdot f_e$, where $|\mathcal{B}|$ is the number of bases, β is the number of elements in the table *locations*, *coding configurations*, and *MIME types* from Figure 6, γ is the number of elements in *File Base table* in Table II, and δ is the number of files in the system which generate a total of N chunks. Thus, the total storage usage is

$$|\mathbb{B}|(k + m_f + b) + \beta \cdot (l + cc + m_t) + \gamma \cdot f_b + \delta \cdot f_e$$

VI. EXPERIMENT RESULTS

We evaluate Alexandria's proof-of-concept against ZFS with deduplication with a standard chunk size of 128 kB [18]. We use EXT4 as a baseline. We use three data sets to evaluate the storage gains of Alexandria. All image data sets contain images of varying size and resolution. We evaluate Alexandria's performance with chunk sizes of 1 and 4 kB and show the current costs as well as the expected (optimised) costs when using a registry that is comparable to ZFS, i.e., implemented at the file system level, which will not require information about MIME Type, basis locations, file size.

TIFF Image Data Set: This data set consists of pan-sharpened images in TIFF format [19] with a total size of 3325 MB, where each file contains multiple images [20].

Table	Size
Base (b)	32
Location (l)	259
File base (f_b)	15
Coding configuration (cc)	259
MIME Type (m_t)	259
File entry (f_e)	275

TABLE II
SIZE OF A ROW ENTRY IN BYTES FOR EACH REGISTRY TABLE

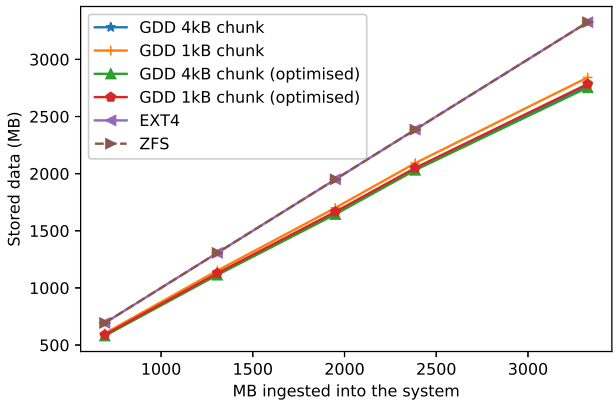

Fig. 7. Results for TIFF image with basis chunk size of 4 kB and 1 kB

Figure 7 shows that ZFS provides less than 1% compression gain, while Alexandria's gain is much more significant. Even when only 693 MB have been stored in the system, we see compression gains of 15.2% and 13.25% using basis chunk sizes of 4 kB and 1 kB, respectively. These compression gains go up to 16.75% (4 kB chunks) and 14.5% (1 kB chunks) when the full 3325 MB have been stored in the system. This clearly shows the benefit of mapping multiple chunks of data to the same basis compared to mapping only identical data as in ZFS' classical deduplication algorithm.

PNG Grey Scale Data Set: This is a collection of grey scale photos in PNG format [21] with a total size of 2789 MB [22], [23]. For the x-ray data set, we see varying results as illustrated in Figure 8. First, ZFS provides an low compression gain ($< 1\%$) compared to Alexandria. We suspect this is due to the bigger chunk size of ZFS, which increases the difficulty of finding identical blocks. The compression gain of Alexandria when all data is stored in the system is 3.5% and 3% for 4 kB and 1 kB chunks, respectively. We see gains of up to 14.2% and 12.44% when storing the first 1240 MB for 4 kB and 1 kB chunks, respectively. This is related to new data coming into the system that could not be mapped to an already registered basis. From a theoretical and practical perspective, we expect much larger gains as sufficient data is stored in the system. Our current result shows that Alexandria can provide compression gains where ZFS cannot, in particular, gains for already compressed data in the form of PNG images.

RAW Image Data Set: This is a collection of images of a different object in RAW format (no data compression), with a total data size of 1904 MB [24]. Alexandria uses the same amount of storage as EXT4 for 4 kB chunks, while for 1 kB

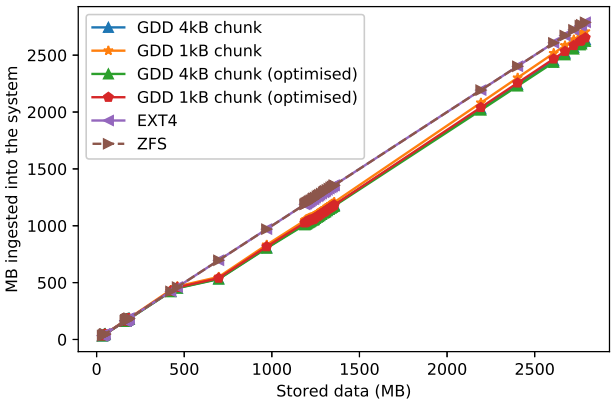

Fig. 8. Result for PNG images with basis chunk size of 4 kB and 1 kB

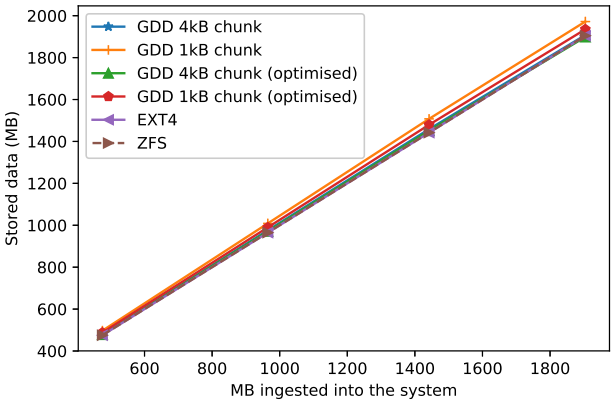

Fig. 9. Result for RAW images with basis chunk size of 4 kB and 1 kB

chunks we see increased storage usage of around 3%. The storage gain for Alexandria is being reduced due to the number of entries in the registry and the need to track additional information, e.g., basis location, compared to ZFS who does not have a similar requirement. This requirement in Alexandria can be removed when considering the optimal registry for a file system that is comparable to ZFS. In this case, the difference between Alexandria, ZFS and EXT4 is negligible.

VII. Conclusion

We presented an architecture supporting Generalised Data Deduplication in Cloud and Edge storage systems and implemented a proof-of-concept of it. We evaluated the implementation using multiple data sets and compared it with the state-of-the deduplication file system, ZFS, and EXT4 as a baseline for standard file systems without deduplication. Our evaluation shows that the uniqueness of the data does affect the performance of GDD but that GDD can provide 16.75% compression gain compared to ZFS even for small data sets. We expect that the gains will increase with larger data sets. Additionally, we have shown that there is room for optimisation of the registry size, which we will address in our future work. Beyond our proposed architecture, future work will consider using GDD for (i) developing file systems, and (ii) standard file compression, due to GDD's potential to allow random access to data, as well as studying the potential of other ECCs for GDD to improve compression gains.

References

[1] J. R. David Reinsel, John Gantz. (2018, 11) The digitization of the world from edge to core. [Online]. Available: https://www.seagate.com/files/www-content/our-story/trends/files/idc-seagate-dataage-whitepaper.pdf

[2] A. T. Clements, I. Ahmad, M. Vilayannur, J. Li *et al.*, "Decentralized Deduplication in SAN Cluster File Systems," in *USENIX annual technical conference*, 2009, pp. 101–114.

[3] R. Kaur, I. Chana, and J. Bhattacharya, "Data deduplication techniques for efficient cloud storage management: a systematic review," *The Journal of Supercomputing*, vol. 74, no. 5, pp. 2035–2085, 2018.

[4] W. Xia, H. Jiang, D. Feng, F. Douglis, P. Shilane, Y. Hua, M. Fu, Y. Zhang, and Y. Zhou, "A Comprehensive Study of the Past, Present, and Future of Data Deduplication," *Proceedings of the IEEE*, vol. 104, no. 9, pp. 1681–1710, 2016.

[5] N. Mandagere, P. Zhou, M. A. Smith, and S. Uttamchandani, "Demystifying data deduplication," in *ACM/IFIP/USENIX Middleware '08 Conference Companion*, ser. Companion '08. New York, NY, USA: ACM, 2008, pp. 12–17.

[6] C. Gonzalez. (2013) ZFS: To Dedupe or not to Dedupe. [Online]. Available: https://constantin.glez.de/2011/07/27/zfs-to-dedupe-or-not-dedupe

[7] R. Vestergaard, Q. Zhang, and D. E. Lucani, "Generalized Deduplication: Bounds, Convergence, and Asymptotic Properties," pp. 1–14, 2019. [Online]. Available: http://arxiv.org/abs/1901.02720

[8] Oracle, "What is zfs?" https://docs.oracle.com/cd/E23823_01/html/819-5461/zfsover-2.html#gayou, 2019, visited: 31/01-2019.

[9] R. W. Hamming, "Error detecting and error correcting codes," *The Bell System Technical Journal*, vol. 29, no. 2, pp. 147–160, April 1950.

[10] json.org, "Introducing json," https://json.org/, 2019, visited: 27/02-2019.

[11] cppreference.com, "Standard library header ¡filesystem¿," https://en.cppreference.com/w/cpp/header/filesystem, 2019, visited: 27/02-2019.

[12] The PostgreSQL Global Development Group, "Postgressql: The world's most advanced open source relational database," http://postgresql.org/, 2019, visited: 30/01-2019.

[13] W. Dai, "Crypto++ library 8.2," https://cryptopp.com/, 2019, visited: 14/06-2019.

[14] The PostgreSQL Global Development Group, "Constraints," https://www.postgresql.org/docs/8.1/ddl-constraints.html, 2019, visited: 14/05-2019.

[15] ——, "Numeric types," https://www.postgresql.org/docs/11/datatype-numeric.html, 2019, visited: 30/01-2019.

[16] ——, "Binary Data Types," https://www.postgresql.org/docs/11/datatype-binary.html, 2019, visited: 30/01-2019.

[17] ——, "Character types," https://www.postgresql.org/docs/11/datatype-character.html, 2019, visited: 30/01-2019.

[18] OpenZFS, "Performance tuning," http://open-zfs.org/wiki/Performance_tuning#Dataset_recordsize, 2019, visited: 19/06-2019.

[19] Adobe Developers Association, "Tiffrevision 6.0final — june 3, 1992," https://www.adobe.io/content/dam/udp/en/open/standards/tiff/TIFF6.pdf, 2019, visited: 17/06-2019.

[20] Planet Labs Inc, "Download samples of our, high resolution imagery, for monitoring, tasking and large area mapping," https://info.planet.com/download-free-high-resolution-skysat-image-samples/, 2019, visited: 17/06-2019.

[21] T. Boutell, "Png (portable network graphics) specification version 1.0," https://tools.ietf.org/html/rfc2083, RFC Editor, RFC 2083, March 1997. [Online]. Available: https://tools.ietf.org/html/rfc2083

[22] D. Mery, V. Riffo, U. Zscherpel, G. Mondragón, I. Lillo, I. Zuccar, H. Lobel, and M. Carrasco, "GDXray: The Database of X-ray Images for Nondestructive Testing," *J. of Nondestructive Eval.*, vol. 34, no. 4, p. 42, nov 2015.

[23] D. Mery, "Gdxray: X-ray images for x-ray testing and computer vision," http://dmery.ing.puc.cl/index.php/material/gdxray/, 2019, visited: 17/06-2019.

[24] D.-T. Dang-Nguyen, C. Pasquini, V. Conotter, and G. Boato, "Raise: A raw images dataset for digital image forensics," in *ACM Multimedia Systems Conf.*, ser. MMSys '15. New York, NY, USA: ACM, 2015, pp. 219–224.