DDFlasks: Deduplicated Very Large Scale Data Store

Francisco Maia, João Paulo, Fábio Coelho, Francisco Neves, José Pereira, Rui Oliveira

DDFLASKS: deduplicated very large scale data store

F. Maia[1], J. Paulo[2], F. Coelho[3], F. Neves[1], J. Pereira, and R. Oliveira

HASLab, INESC TEC & U. Minho, Braga, Portugal
{fmaia, jtpaulo}@di.uminho.pt,
{fabio.a.coelho,francisco.t.neves}@inesctec.pt, {jop,
rco}@di.uminho.pt

Abstract. With the increasing number of connected devices, it becomes essential to find novel data management solutions that can leverage their computational and storage capabilities. However, developing very large scale data management systems requires tackling a number of interesting distributed systems challenges, namely continuous failures and high levels of node churn. In this context, epidemic-based protocols proved suitable and effective and have been successfully used to build DATAFLASKS, an epidemic data store for massive scale systems. Ensuring resiliency in this data store comes with a significant cost in storage resources and network bandwidth consumption. Deduplication has proven to be an efficient technique to reduce both costs but, applying it to a large-scale distributed storage system is not a trivial task. In fact, achieving significant space-savings without compromising the resiliency and decentralized design of these storage systems is a relevant research challenge.
In this paper, we extend DATAFLASKS with deduplication to design DDFLASKS. This system is evaluated in a real world scenario using Wikipedia snapshots, and the results are twofold. We show that deduplication is able to decrease storage consumption up to 63% and decrease network bandwidth consumption by up to 20%, while maintaining a fully-decentralized and resilient design.

1 Introduction

For many years now we hear promises of the emergence of the Internet of Things (IoT) and of Edge Computing. Still, the world of interconnected things has remained more an idea than a concrete reality. Recent predictions from the International Data Corporation (IDC) studies, however, point to significant developments in this area and it is expected that by 2020 there will be an extraordinary number of 32 billion things connected to the Internet [14]. Moreover, the amount of digital data will grow from 4.4 Zettabytes in 2013 to 44 Zettabytes in 2020.

Naturally, an explosion in the number of connected devices and in the amount of data being produced and exchanged demands for novel approaches to data management. Massive scale systems, composed of thousands to millions of devices, exhibit specific characteristics that are specially challenging and need to be addressed. Namely, the increase in scale is necessarily accompanied by an increase in system dynamism. Such dynamism arises both from failures that, in these environments, become the rule instead of the exception and by the natural constant entrance and departure of devices, which we will call nodes from now on.

Alongside, real world applications start to struggle to find affordable systems to manage and store massive amounts of data. As an example, the Wikimedia Foundation is currently requesting help to users that have spare storage and bandwidth capabilities to store and host Wikipedia snapshots[1]. These snapshots contain the entire history of Wikipedia across distinct periods of time and are valuable for a wide variety of users including researchers. However, they are not easily accessible due to limited storage capabilities. Thus, offering a massive scale storage system able to accommodate the entire Wikipedia and its history relying only on commodity hardware becomes of significant interest. Moreover, serving all these snapshots from an unified storage service, instead of scattering the snapshots across independent storage systems, is key for users to have an efficient way of accessing the full history of Wikipedia.

Recent research work proposed a data store entirely built with epidemic protocols, tailored precisely for large scale environments [18]. The success of DATAFLASKS, with respect to coping with high levels of system dynamism, lies in its autonomous and unstructured approach to node organization and in its pro-active approach to fault tolerance. In DATAFLASKS, nodes autonomously organize themselves into groups that are responsible for a subset/partition of the data. Then, the number of nodes in a group determines the data replication factor for the data being stored. The effectiveness of a pro-active approach to data replication comes, unfortunately, with an increase in storage and network resource usage. In fact, bandwidth is actually a bottleneck for scalability in this type of systems and, even though DATAFLASKS autonomous data partitioning alleviates the problem, this still weakens its applicability in real world scenarios [2]. Alongside, as all nodes belonging to the same group are fully-replicated, the available storage space provided by the group is limited to the size manageable by the single node with the lesser storage capabilities. This restriction is of special importance if we consider each node to be commodity hardware or even smaller edge devices where storage space available is limited.

Data deduplication has proven to be an efficient technique for finding and eliminating duplicate content in large volumes of data [21]. Moreover, it was used in the past to reduce the network bandwidth consumption of distributed storage systems. However, leveraging deduplication in a massive-scale data store such as DATAFLASKS is not a trivial task. One approach is to apply local deduplication only for the data stored in each node. As this approach does not eliminate duplicates stored across distinct nodes, it requires an efficient content-aware policy for distributing data to nodes that maximizes the obtainable space-savings. Other approach is to perform global deduplication across data stored in all nodes, thus finding redundancy across the entire storage system. However, finding duplicates across all nodes requires global metadata and coordination, which not only increases the complexity of the system but may also compromise the decentralization, fault-tolerance, and performance of systems such as DATAFLASKS.

Contributions We propose DDFLASKS, a massive scale deduplicated data store. It shows the applicability of integrating DATAFLASKS, a massive scale data store, with deduplication, without loosing any of its design guarantees, such as decentralization and high churn tolerance. Additionally, we evaluate its effectiveness using a real workload, specifically storing and serving simultaneously both the most recent versions of

[1] `https://dumps.wikimedia.org`

Wikipedia [10] articles and their older historical versions. In fact, using real data from Wikipedia, we show that our system is able to store and serve articles across several nodes with high levels of storage savings up to 63% and network savings up to 20%.

Roadmap The rest of the paper is organized as follows. In Section 2 we describe the architecture and design of DATAFLASKS, the baseline system used to build our novel approach. Next, in Section 3 we describe the Wikipedia use case and present some preliminary results that motivate the usage of deduplication. In Section 4 we introduce DDFLASKS. We then proceed to DDFLASKS evaluation in Section 5 and present related work in Section 6. The paper is concluded in Section 7.

2 DATAFLASKS: Epidemic store for massive scale systems

The pivotal idea guiding the design of DATAFLASKS is decentralization, where each node is autonomous and all nodes play the same role [18]. A node progresses relying solely on local decisions without depending on any other node and on any kind of hierarchy. When a client issues a request, such request is disseminated throughout the system and each node decides how to handle it. Store requests are composed by an identifier of the object to be stored that must be unique, by the version of the object to be stored, and by the object's data. Storing several versions of the same object is important for many applications that resort to data versioning.

Briefly, the API is composed by a *get* and *put* operation. When a *get* is received, if the node holds the corresponding triple (key,version,object) it replies to the client. Otherwise, it ignores the request. In the case of a *put* operation, the node locally decides to store the corresponding triple (key,version,object) or to discard it. The decision to store or not the data is used to implement data distribution and replication. DATAFLASKS ensures that a sufficient number of nodes actually decides to store each data object in order to guarantee data replication, and thus, to tolerate node failures.

The set of nodes that takes the same decisions on whether to store data objects or not is viewed as a group. Accordingly, the decision of which data to store is reduced to the decision of which group a node belongs to. Once that decision is made, each node is responsible for a subset of the data according to a deterministic mapping between the pair (key,version) of an object and the group it belongs to. Data is thus distributed by groups, providing load balancing, and replicated a number of times equal to the size of the group. Strikingly, each node is able to decide to which group it belongs without requiring any kind of coordination.

In order to achieve this, the system is entirely built with unstructured and pro-active epidemic protocols. They are characterized by their independence from any kind of structure or hierarchy among nodes and by the fact that they rely on pro-active mechanisms for fault tolerance that are able to antecipate system repair. The result is a completely decentralized and coordination-free data store. Characteristics that make DATAFLASKS inherently scalable and able to cope with unprecedented levels of system dynamism, may it be caused by membership instability or by failures.

In the system's architecture, each node runs five components: *Membership*, *Group Construction*, *Storage*, *Replica Maintenance* and *Interface*. In order to provide some background and context to the design of the system proposed in this paper, we briefly describe how each component works in the original setting.

The *Membership* component is responsible for providing each node with a list of available nodes in the system. It does so guaranteeing that such list represents a random sample of nodes from the entire system and that it is periodically refreshed. It is important to notice that each membership list is always a small subset of nodes with respect to the system size, which allows the system to scale.

The *Group Construction* This component is responsible for determining to which group the node belongs. As described previously, the group determines which data to locally store or to discard. Without going into much detail, this component works by leveraging information being propagated at the membership level to estimate the number of groups needed to satisfy a desired, user defined, replication factor. Then, the node places himself on one of those groups guaranteeing that system nodes are uniformly distributed across the different groups. For a detailed description of the protocol please refer to [18]. Once in a group, each time a *put* operation is issued for a certain key, that key is mapped deterministically to a group by using an hash function. As described further on, this mapping allows different versions of the same key to be placed in the same replication group. This will allow maximizing deduplication effectiveness.

The *Storage* component abstracts the actual medium to which the data is persisted. Currently, this component can be configured to be a in-memory store or a disk-based one. This paper introduces a new storage component to support data deduplication.

In order to maintain the replication level in the presence of churn, the *Replica Maintenance* component periodically publishes to other nodes in the group the set of keys it currently holds locally. Within a group, all nodes store the same set of data objects. Upon receiving a maintenance message, each node checks if it is storing all keys correspondent to the group. If not, it requests the missing data from the nodes in its group. In this paper we provide a new replica maintenance component which allows to optimize this process by avoiding to transmit duplicate data through the network.

Finally, the *Interface* component is responsible for handling the incoming connections from other nodes and managing the request workflow in the system. In order to issue *put* or *get* requests the client only needs to be able to contact a single node in the system. The request is then forward appropriately to the correct nodes that can fulfill it.

3 Duplicates in the real world

Many large information systems tend to exhibit a significant amount of duplicate data [19]. This is particularly true for storage systems that evolve incrementally with time. A paradigmatic example is Wikipedia, also known as the Internet encyclopedia [10]. The Wikipedia allows users to create and complement articles about virtually any subject. Articles evolve through time and periodic snapshots of the entire Wikipedia are stored for future reference. Because Wikipedia serves a very high volume of requests and stores a growing large volume of data, it is a suitable use case for DATAFLASKS that can leverage its highly scalable infrastructure to serve Wikipedia's high demand.

Naturally, different versions of the same article share significant portions of the text, which is redundant when stored. This means that a storage system holding the full history of Wikipedia is expected to have a considerable amount of duplicate content [11]. A possible approach to eliminate this redundancy would be to use a traditional compression technique such as gzip. However, compression techniques are ideally designed

Table 1. Analysis of duplicates results with 1024, 2048 and 4096 bytes Rabin fingerprints for a single group of the DataFlasks configuration with 40 groups.

Fingerprint Avg Size	# articles	Total space (GB)	total # blocks	# unique blocks	# duplicate blocks	Avg # copies / duplicated block	Space saved (GB)	duplicate space %
1024	1,393,130	7.63	7,046,744	4,226,205	2,820,539	3.20	3.27	42.88
2048	1,393,130	7.63	3,995,416	2,870,780	1,124,636	2.59	2.59	33.99
4096	1,393,130	7.63	2,550,938	2,132,849	418,089	2.65	1.89	24.81

to eliminate intra-file redundancy or redundancy over a small group of files, typically stored together in the same operation. In the Wikipedia use-case, new versions of the same article are created over time and must be retrieved efficiently if requested. This means compressing and decompressing data several times which results in a significant penalty on storage requests performance. Another possible approach to eliminate such redundancy and to spare storage space is to use incremental backup techniques such as delta-encoding. With this technique new versions of a previously stored article are stored as deltas or *diffs* that only contain the content that was actually modified. These deltas can then be applied to the original (base) article to rebuild a specific version of the article. Although this technique is efficient in terms of storage space savings, it requires additional computational power and it is slower than deduplication, specially when articles have a large number of versions and several deltas must be applied to the base article to retrieve latest versions. For this reason, this paper proposes the use of block-based deduplication, which allows users to query any article version in the past and get the response without the need to rebuild a set of deltas or decompress data [21].

To validate that deduplication is, in fact, suitable and effective for a deployment where DATAFLASKS is serving Wikipedia articles, we performed the following experiment. We used 15 monthly Wikipedia snapshots taken for the period between November of 2014 and January of 2016[1]. Each snapshot has the latest full version of all articles belonging to the English version of Wikipedia. The snapshots were processed by the order they were taken and the corresponding articles were stored in a way that mimics the distributed storage approach taken by DATAFLASKS in a real deployment *i.e.*, articles were divided into groups and stored accordingly. Each group of articles represents the data partition that would be assigned to a specific set of DATAFLASKS nodes. We then focus our analysis on each one of the partitions. It is important to notice that deduplication will be applied locally by each node. Consequently, nodes in the same group, that replicate the same data partition, will store the same content, which makes it sufficient to analyze a single node per group. Additionally, across consecutive snapshots there are some repeated articles that remained unmodified and were not stored in our experiment.

On the other hand, new versions of previously stored articles were routed to the same data group, where their ancestors were persisted, and were stored as new objects (files) with distinct version identifiers. This way, the experiments stored the full content for each article version which is in conformity with the rationale explained previously where our very large data store is used to serve several articles and their distinct versions without requiring the usage of incremental backup techniques.

After populating the distinct data groups with the Wikipedia dataset the global storage space in use was $\approx$ 305 GB, corresponding to 55,745,648 articles. In order to check the percentage of redundancy in the stored dataset, we resort to the DUPSANALYSER tool an open-source project (`https://github.com/jtpaulo/dupsanalyzer`) that processes the content of files and extracts statistics for the duplicate content found. Duplicates can be found either by searching for duplicate blocks with a fixed or variable size. The latter resorts to an implementation of the Rabin Fingerprint scheme for calculating variable-sized blocks and their corresponding content hashes efficiently [20]. As Wikipedia articles are text articles, using variable sized blocks is a better choice for finding duplicates [11, 21]. Briefly, lets consider two versions of the same article where version A only differs from version B by a single character that was added to the beginning of the latter version. If the two articles are scanned with a fixed size partitioning scheme, no blocks from version A will match blocks from version B. In contrast, the Rabin fingerprint scheme uses a sliding window that moves through the data until a fixed content pattern defining the block boundary is found. This approach generates variable-sized blocks and solves the issue of inserting a single byte in the beginning of version B. More precisely, only the first block from version B will differ from the first block of version A due to the byte addition, while the remaining blocks will still be duplicate. Finally, the Rabin scheme is configurable with target average, maximum and minimum block size, which allows avoiding the generation of very small or large blocks while still keeping their sizes variable. In the results discussed next, we used DUPSANALYSER to process the articles, and corresponding versions, stored at each data group. Individually, for each data group, our analysis tool processed all stored files to find intra and inter-file duplicates.

Distinct group sizes results. Our first experiment was designed to check the amount of duplicates found per group node when dividing articles into 10, 20 and 40 groups for different block sizes: 1024, 2048 and 4096 bytes. With 10 groups each group node holds $\approx$ 30GB, with 20 groups $\approx$ 15GB and with 40 groups $\approx$ 7.5GB. We noticed that the percentage of duplicates found does not increase significantly if a group holds more data, because most redundancy is originated by storing distinct versions of the same article in the same group, which happens identically for the three group sizes.

Single group analysis for the 40 groups scenario. Since the percentage of duplicates does not change significantly when considering different number of groups, we show in Table 1 a more detailed analysis of the stored content in a single group for the experiment with 40 groups. The analyzed group holds 7.63 GB of data corresponding to more than one million articles. For each Rabin fingerprint size, the total number of generated blocks diverges and, as expected, with a smaller size it is possible to find more duplicates and have significantly higher space savings. However, reducing the block size increases the size of the metadata used to index all stored blocks and to find duplicates.

To conclude, these results show that single-node deduplication with a variable 1024 bytes fingerprinting scheme allows reducing 45% of the storage space occupied by 15 snapshots of the English Wikipedia version.

4 DDFLASKS

Recalling Section 2, data distribution and replication in DATAFLASKS is achieved by dividing nodes into groups. Each group is responsible for a set of data and, accordingly,

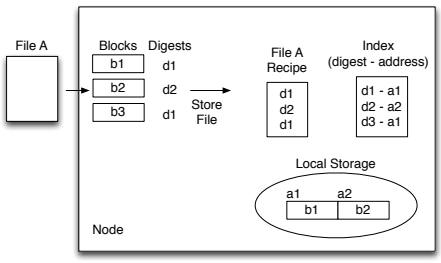

Fig. 1. Deduplication in DDFLASKS

each node belonging to that group will have to store that specific set of data in its local storage. The Wikipedia study discussed in the previous section shows that a significant percentage of duplicates exists in each node when all the versions of a specific article are grouped together. In DDFLASKS, this insight is leveraged by ensuring that data objects identified by a key are always assigned to the same group independently of their version. With this approach, all the versions of an article are stored in the same group while clients can still retrieve specific versions of an article by specifying the article's key and the desired version. This is achieved by taking into advantage the load balancing mechanism from the original DATAFLASKS, which deterministically routes a certain key to a group. DDFLASKS inherits characteristics from DATAFLASKS, such as fully-decentralization. In particular, it resorts to node-local deduplication that does not require any global index or coordination mechanisms that would impact high-churn tolerance and the performance of storage requests [21].

In comparison with the baseline architecture discussed in Section 2, DDFLASKS is extended with storage and network deduplication mechanisms. The resulting open-source system is available at `http://github.com/fmaia/dataflasks`.

First, a new storage component is provided with integrated in-line local storage deduplication, which works as follows. In each node, duplicates are identified and eliminated before actually being stored persistently. In the literature this approach is known as in-line deduplication [21]. Duplicates are found by resorting to an *index* that maps blocks with unique content to their respective storage addresses. When a block is being written, a digest of the block's content is calculated and the index is searched for a possible duplicate. If a duplicate exists, then the new block does not need to be stored, otherwise, the block is stored and the index is updated with a new entry for that block. A Rabin Fingerprint scheme identical to the one described in Section 3 is used to divide files into variable-size blocks and to calculate small digests of their content [20]. This way, the index does not store the actual block but a smaller digest identifying the content of that block. Fingerprints are deterministically calculated per-file. Thus, at each node, storing files in different orders does not affect the correctness of the approach. In order to retrieve files from the storage system, an additional metadata structure, that we refer to as *file recipe* is used. Each file recipe identifies a single file stored on DDFLASKS and tracks the digests of the blocks that belong to that specific file. The actual storage address of these digests can be consulted at the index. Deduplication is thus achieved because file recipes with duplicate content share digests that are mapped to the same storage block. Figure 1 shows an example of the proposed single-node deduplication mechanism. As the first step, File A is routed to the correct group of nodes. Then, in each node storing the file, the file is divided into variable-sized blocks and a digest for the content of each block is calculated. In the example, block1 and block3 have the

same content. Each digest is checked at the index and if not found, a new entry is added while the corresponding block is stored in a append-only storage. In the figure blocks $b1$ and $b3$ are duplicates, so only block $b1$ and $b2$ are stored. Finally, the file recipe for File A is also kept at the node in order to fetch all the necessary blocks when a client asks for that file. The index keeps the digests and corresponding location for all blocks at the local storage which enables both intra- and inter-file deduplication for all files stored in the same node. In Section 5 we show that our approach is still able to achieve significant storage space savings even when metadata space is accounted for.

In this paper we do not address data deletion functionalities. This is motivated by the fact that DDFLASKS is a large-scale system intended to store large amounts of archival data. For use-cases such as the Wikipedia one used in the paper, this is a practical assumption since the main goal is to keep all versions of wikipedia articles without ever deleting them. As described in the previous section, for this use case, single-node deduplication proves to be an efficient technique to spare redundant storage space and avoids scalability issues found in large-scale in-line deduplication systems that must maintain a global index for finding duplicates across remote storage nodes [7, 8].

The second deduplication mechanism proposed in the paper aims at optimizing the network bandwidth used by DDFLASKS data replication techniques. In order to cope with high levels of node churn and to maintain desirable data replication levels, each system node proactively and periodically contacts other nodes in the same group to announce the set of files it is currently storing. If one node receives this set and verifies that its local storage is currently missing some files, it must contact other nodes in the same group to ask for those files. Naturally, when churn levels become significantly high, the volume of data traversing the network increases as more files are being exchanged. We propose to mitigate this problem by employing deduplication to the data being exchanged between nodes. In detail, nodes periodically announce to the group not only the set of files that they currently hold but also the digests that compose those files. When a node receives this list and verifies that a set of files is missing, it checks first what digests from those files are already stored locally. This can be done by leveraging the index metadata used for local storage deduplication. Then, the node only requests the blocks that are actually missing in its local storage. After receiving these blocks the node updates the index and creates the corresponding file recipes. A key advantage of this mechanism is that it relies on the metadata already used for performing in-line deduplication, which is an idea that has proven successful in previous proposals for backing up data across peer-to-peer networks [5, 20]. Although this strategy requires sending the list of digests when announcing the files that nodes currently hold, we show in Section 5 that it still spares significant network bandwidth. Note that although single-node deduplication is already provided in several storage appliances, it is not trivial to incorporate these solutions with DDFLASKS and take advantage of the deduplication metadata, that is in most cases is protected within the appliance, to implement the previous network optimizations.

Implementation details The two deduplication mechanisms were implemented on top of the current implementation of the system described in Section 2. The deduplication index is an in-memory HashMap that maps blocks digests (8 bytes) to storage addresses (8 bytes)[2]. Similarly, file recipes are stored in an in-memory HashMap that

maps the identifier of a file (16 bytes, 8 bytes for the file key and 8 bytes for the version) to its file recipe whose size depends on the number of block digests composing that file. DDFLASKS is mainly thought for running in commodity hardware nodes and the amount of data hold by each node is not expected to be very large (tens to hundreds of GBs). So, the amount of metadata held by each node is also expected not grow to large values. Additionally, in the context of this paper we assume that, even in the presence of high levels of churn, for each group there is always a set of live nodes. This way metadata for freshly booted nodes can always be reconstructed from live nodes.

5 Evaluation

DDFLASKS was evaluated in a real deployment to validate two main claims. First, that deduplication allows sparing significant storage space for each node. Second, that the network bandwidth used by nodes when exchanging messages is also reduced.

To this end, we have performed a set of experiments that demonstrate the effectiveness of the deduplication mechanism implemented. Each experiment was run both in the original DATAFLASKS, non-deduplicated system (used as the baseline) and in DDFLASKS. The experiment set up consists of a cluster of commodity hardware nodes equipped either with a 3.1 GHz Dual-Core Intel i3 Processor, 8 GB of RAM and a 7200 RPMs SATA disk or a 3.7 GHz Dual-Core Intel i3 Processor, 8 GB of RAM and a SSD disk. All nodes are connected through a gigabit ethernet switch. It is important to notice that hardware heterogeneity does not impact the results of our experiments. In fact, it is out of the scope of the present paper the evaluation of system performance metrics. These metrics will mostly be affected by the deduplication approach being used *i.e.,* fingerprinting scheme, index scheme, etc. As discussed in previous work, each scheme adds different tradeoffs in terms of storage performance, deduplication performance and resources (RAM, CPU, Disk) consumption [21].

Instead, we focus on analyzing storage and network savings achievable by our system. Similarly, the validation of DDFLASKS scalability to thousands of nodes and resiliency to high churn rations is already addressed in previous work [18].

Leveraging the results obtained in Section 3 and aiming at real world assessment of DDFLASKS, all the experiments presented next resort to actual Wikipedia data.

5.1 Storage Savings

In order to evaluate the storage behavior of DDFLASKS we have considered 15 Wikipedia monthly snapshots. Each one of these snapshots contains a set of articles from the English version of the Wikipedia. From snapshot to snapshot each article may change reflecting its evolution through time. In the real world deployment of Wikipedia, users see only a single (latest) snapshot. However, in our scenario we want to go a step forward and it is our goal to simultaneously store and serve several Wikipedia snapshots.

The 15 snapshots used amount to $\approx$ 115 GB corresponding to $\approx$ 6.3 million articles. Each article is stored as a single data object in the storage system and each new article

[2] For each entry at the index, 4 extra bytes must be stored because variable sized blocks are being used and their size must also be kept.

Table 2. Storage and metadata space occupied for DDFLASKS and the DATAFLASKS storage systems

	DATAFLASKS	DDFLASKS
Global storage space (GB)	115.5	42.4
Average storage space / node (GB)	7.2 ($\pm$ 0.08)	2.65 ($\pm$ 0.05)
Global Deduplication savings (GB)	-	73.1
Average deduplication Savings / node (GB)	-	4.55
Global Metadata space (GB)	1.32	12.04
Metadata space / node (GB)	0.08 ($\pm$ 0.003)	0.75 ($\pm$ 0.05)

snapshot corresponds to a new version of such object. Moreover, article versions are treated as new articles thus identified with the same key as the original article but with a different version number. This information is used by DDFLASKS to collocate articles with their subsequent versions in the same node group.

We configured both DATAFLASKS and DDFLASKS to arrange nodes into 16 groups. Each group is responsible for storing a subset of the articles written to the store. As described previously, all nodes belonging to a certain group store the same data and deduplication is applied locally to each node. Consequently, in order to observe the system's behavior it is sufficient to analyze the behavior of a single node per group. Other nodes in the same group will exhibit exactly the same results as the ones presented next.

The experiment consisted on loading both DATAFLASKS and DDFLASKS with the 15 data snapshots writing each article and subsequent versions in chronological order (from the oldest snapshot to the latest one). After the load was completed we analyzed the storage usage of a node per group.

In Table 2 we present the results of this experiment. It is observable that DDFLASKS is significantly more frugal than DATAFLASKS with respect to storage space usage. The former requires 42.4 GB to store all the articles while the latter, without deduplication, requires 115.5 GB. In detail, 73.1 GB are saved by using deduplication which corresponds to a space saving of 63% when compared to the baseline approach. Please note that, when compared with the motivation tests described in Section 3, there is an improvement in the storage savings results. This improvement is explained by the fact that, in this real deployment, we used a sample of the articles (and corresponding versions) used in the motivation experiments, which happen to exhibit slightly higher redundancy between them. Additionally, we can observe that the local storage space required by nodes in different groups is similar and that the deduplication savings in each node are identical to the one observed globally for the whole storage by considering a load balancing strategy that routes articles uniformly across distinct groups.

Going into some detail, we also show in the table the space used by metadata structures. In both systems, more than 390,000 articles were stored in each node. As expected, deduplication requires additional metadata space for storing and indexing articles' blocks, while in the baseline system it is only required a simpler file recipe that points a specific file to its storage address. Nevertheless, the space savings achieved clearly compensate the overhead introduced by the extra metadata structures used in DDFLASKS. In fact, less than 17% of the space spared by deduplication is needed for fulfilling the extra metadata space overhead. Finally, Table 3 shows the exact space oc-

Table 3. Space occupied by DDFLASKS index and file recipe

Metadata	Global space (GB)	Space / node (GB)
Index	5.35	0.33 ($\pm$ 0.002)
File recipe	6.69	0.42 ($\pm$ 0.003)

cupied by the index and file recipe metadata in our system. Again, the space occupied by each metadata structure across different nodes does not change significantly.

5.2 Network Savings

Replication is achieved in our system resorting to periodic message exchanges between nodes with information about the data objects they are storing. Each time, following a message exchange, a node detects it is missing some object it requests it from other nodes in the same group. Naturally, if the system is stable, it is expected that nodes store all correspondent data objects and that these message exchanges do not yield missing data requests. However, when nodes fail or enter the system data objects need to be requested to maintain the desirable replication levels.

In this experiment, we show that deduplication can reduce network consumption of the data exchange mechanism between nodes. We focus on two nodes belonging to the same group and observe their behavior when one of them keeps failing and re-entering the system while the system is continuously being loaded with new data. Naturally, it is expected that each time the node re-enters the system it will request missing data from its peer that runs continuously. The test ran for 2 hours and after the first 30 minutes one of the nodes was stopped in intervals of 20 minutes. In detail, after being stopped the node remains offline for 20 minutes and then it is rebooted again and it is kept online for additional 20 minutes. This cycle was repeated until the last 30 minutes of the test when the two nodes were kept online. The node being stopped saved its metadata to disk periodically to ensure that when rebooted the index and file recipe metadata were holding previously stored information. Again, 15 Wikipedia monthly snapshots were used, and both systems (DDFLASKS and baseline) stored more than 400,000 articles, which corresponds to $\approx$8.3GB. Please recall that the two nodes were configured to be in the same group so these were fully-replicated, each holding the same amount of articles mentioned previously. In terms of storage space savings the DDFLASKS nodes stored 4.3 GB while the baseline system nodes stored 8.3 GB. This corresponds to a space saving of $\approx$49%, which is in conformity with the results discussed previously and in Section 3. The metadata space required by each node is also compensated by the space savings as in the previous results.

The baseline approach, without network deduplication, sends more than 22GB through the network while the deduplication approach only sends 17.71GB. Note that these bandwidth consumption results consider all network traffic. In fact, while most of this traffic is due to the data replication mechanism, system control traffic and client requests are also accounted for in the total value. Moreover, both systems rely on the UDP protocol that requires resending messages that are lost due to failures of the protocol, which also increases network bandwidth usage. Nevertheless, these results show that only by using deduplication for the data replication mechanism it is possible to spare $\approx$20% of all the data exchanged across replicated peers.

The previous results show that significant storage space and network bandwidth can be spared with DDFLASKS. We expect these savings to be similar for other backup workloads with periodic snapshots. In fact, as presented in [19], some of these backup workloads will have higher duplication ratios than Wikipedia, meaning that the network and storage savings achievable should also be higher.

6 Related work

In the pursuit for large scale data management, traditional relational database systems have been, for certain domains and applications, largely replaced by new approaches to data management. Commonly know as NoSQL data stores, these data management systems offer relaxed consistency guarantees when compared with traditional relational database management systems. Examples are Dynamo, PNuts, Bigtable, Cassandra and Riak [3, 4, 6, 15, 16]. One of the key features of these data stores is how they implement data distribution and discovery. Leveraging scalability properties of peer-to-peer protocols, all these data stores rely on a distributed hash table (DHT) such as Chord or variants to distribute and locate data objects [24]. The exceptions are Bigtable and PNUTS, which are centrally managed instead and typically use a specific DHT variation called 'one-hop' DHT [13]. This variation allows faster lookups but requires complete membership knowledge, *i.e.*, each node knows about all other nodes in the system. Moreover, DHTs are know to struggle in the presence of high levels of churn [23]. As a result, even if the distributed and peer-to-peer nature of these data stores is closely related to DATAFLASKS, this system presents an unique unstructured and pro-active approach to node organization and data replication.

To our best knowledge, applying deduplication to epidemic massive scale systems for improving the usable storage space of peers and to improve the network bandwidth usage of gossip protocols and pro-active replication mechanisms is a novel contribution of this paper. To achieve these goals, we leverage ideas of previous work on deduplication for distributed storage systems [21]. In more detail, for achieving both storage and network savings, in-line local deduplication is applied so that duplicates are eliminated before being stored persistently [8, 22]. In fact, for sparing network bandwidth, duplicates are eliminated before even being sent through the network [20].

Peer-to-peer in-line deduplication, where backups are made cooperatively with remote nodes, was introduced in Pastiche [5]. In this system, nodes backup their data to other remote nodes that are chosen by their network proximity and data similarity. Only non-duplicate data is sent through the network and since nodes with similar datasets are chosen, the amount of data that must be sent through the network and stored in each peer is reduced significantly. Other distributed deduplication systems propose novel load balancing designs that route similar data to the same node in order to optimize the amount of duplicates found and, consequently, maximize storage space savings. These proposals rely on centralized indexes that have global knowledge of the content stored in all nodes, on distributed indexes that scale better than the centralized ones, on statefull and stateless routing algorithms, and on probabilistic routing algorithms that do not need a global knowledge of the content of each node in the system [1,7–9,11,12,17,25].

Although DDFLASKS could benefit from some of the ideas and optimizations discussed in previous deduplication systems, our current design uses the original load bal-

ancing algorithm proposed by DATAFLASKS. Our approach collocates different versions of the same data objects, which are expected to have duplicated content. Deduplication is thus performed locally on each node *i.e.*, each node manages its own index and only eliminates duplicates that are stored on its local storage. Strikingly, as shown in the paper, for realistic use-cases such as the Wikipedia one, ensuring that the same versions of articles are routed to the same DDFLASKS group is enough to achieve significant storage space savings while keeping metadata overhead acceptable. Additionally, our deduplication design can be leveraged to spare not only storage space but also network bandwidth usage across nodes. For epidemic data stores such as DDFLASKS this is a novel contribution that reduces significantly the number of messages exchanged across nodes, thus improving the efficiency of current gossip protocols, which is of particular importance since bandwidth consumption is critical in these systems [2]. Furthermore, our approach does not impact the decentralization and high-churn tolerance assumptions of the original DATAFLASKS system.

7 Conclusion

This paper describes a deduplicated massive scale data store, which can handle high volumes of data while minimizing storage resource usage. DDFLASKS is built resorting to a stack of proactive and completely decentralized gossip-based protocols.

The core idea driving this store is effective data dissemination and independent, local decisions of what to do with the data at each node. In-line deduplication is employed at each node and we show, resorting to a real world scenario, that the system is able to save up to 63% of storage space, in comparison with a non deduplicated one.

Additionally, DDFLASKS design is completely decentralized and is able to cope with unprecedented amounts of churn, while saving up to 20% in network bandwidth consumption when compared with the original DATAFLASKS non deduplicated system.

Acknowledgments The research leading to these results was part-funded by (1) Project TEC4Growth - Pervasive Intelligence, Enhancers and Proofs of Concept with Industrial Impact/NORTE-01-0145-FEDER-000020 is financed by the North Portugal Regional Operational Programme (NORTE 2020), under the PORTUGAL 2020 Partnership Agreement, and through the European Regional Development Fund (ERDF); (2) the ERDF European Regional Development Fund through the Operational Programme for Competitiveness and Internationalisation - COMPETE 2020 Programme within project POCI-01-0145-FEDER-006961, and by National Funds through the FCT Portuguese Foundation for Science and Technology as part of project UID/EEA/50014/2013 and by (3) the European Union's Horizon 2020 - The EU Framework Programme for Research and Innovation 2014-2020, under grant agreement No. 732051;

References

1. Bhagwat, D., Eshghi, K., Long, D.D.E., Lillibridge, M.: Extreme Binning: Scalable, Parallel Deduplication for Chunk-based File Backup. In: International Symposium on Modelling, Analysis, and Simulation of Computer and Telecommunication Systems. pp. 1–9 (2009)
2. Blake, C., Rodrigues, R.: High availability, scalable storage, dynamic peer networks: Pick two. In: Conference on Hot Topics in Operating Systems - Volume 9. pp. 1–1 (2003)
3. Chang, F., Dean, J., Ghemawat, S., Hsieh, W.C., Wallach, D.A., Burrows, M., Chandra, T., Fikes, A., Gruber, R.E.: Bigtable: A distributed storage system for structured data. ACM Transactions on Computer Systems (TOCS) 26(2), 4 (2008)
4. Cooper, B.F., Ramakrishnan, R., Srivastava, U., Silberstein, A., Bohannon, P., Jacobsen, H.A., Puz, N., Weaver, D., Yerneni, R.: Pnuts: Yahoo!'s hosted data serving platform. VLDB Endowment 1(2), 1277–1288 (2008)

5. Cox, L.P., Murray, C.D., Noble, B.D.: Pastiche: Making Backup Cheap and Easy. In: Symposium on Operating Systems Design and Implementation. pp. 1–13 (2002)
6. DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., Vogels, W.: Dynamo: amazon's highly available key-value store. ACM SIGOPS Operating Systems Review 41(6), 205–220 (2007)
7. Dong, W., Douglis, F., Li, K., Patterson, H., Reddy, S., Shilane, P.: Tradeoffs in Scalable Data Routing for Deduplication Clusters. In: USENIX Conference on File and Storage Technologies. pp. 15–29 (2011)
8. Douceur, J.R., Adya, A., Bolosky, W.J., Simon, D., Theimer, M.: Reclaiming Space from Duplicate Files in a Serverless Distributed File System. Tech. Rep. MSR-TR-2002-30, Microsoft Research (July 2002)
9. Dubnicki, C., Gryz, L., Heldt, L., Kaczmarczyk, M., Kilian, W., Strzelczak, P., Szczepkowski, J., Ungureanu, C., Welnicki, M.: HYDRAstor: a Scalable Secondary Storage. In: USENIX Conference on File and Storage Technologies. pp. 197–210 (2009)
10. Fountation, W.: Wikipedia web page. `https://www.wikipedia.org` (2016)
11. Frey, D., Kermarrec, A.M., Kloudas, K.: Probabilistic deduplication for cluster-based storage systems. In: ACM Symposium on Cloud Computing. pp. 1–14 (2012)
12. Fu, Y., Jiang, H., Xiao, N.: A Scalable Inline Cluster Deduplication Framework for Big Data Protection. In: International Middleware Conference. pp. 354–373 (2012)
13. Gupta, A., Liskov, B., Rodrigues, R.: Efficient routing for peer-to-peer overlays. USENIX Symposium on Networked Systems Design and Implementation (2004)
14. IDC: The digital universe of opportunities: Rich data and the increasing value of the internet of things (April 2014), `http://www.emc.com/leadership/digital-universe/2014iview/executive-summary.htm`
15. Klophaus, R.: Riak core: building distributed applications without shared state. In: ACM SIGPLAN Commercial Users of Functional Programming. p. 14 (2010)
16. Lakshman, A., Malik, P.: Cassandra: a decentralized structured storage system. ACM SIGOPS Operating Systems Review 44(2), 35–40 (2010)
17. Lillibridge, M., Eshghi, K., Bhagwat, D., Deolalikar, V., Trezise, G., Camble, P.: Sparse Indexing: Large Scale, Inline Deduplication Using Sampling and Locality. In: USENIX Conference on File and Storage Technologies. pp. 111–123 (2009)
18. Maia, F., Matos, M., Vilaça, R., Pereira, J., Oliveira, R., Rivire, E.: Dataflasks: Epidemic store for massive scale systems. In: International Symposium on Reliable Distributed Systems. pp. 79–88 (2014)
19. Meyer, D.T., Bolosky, W.J.: A study of practical deduplication. ACM Transactions on Storage 7(4) (2012)
20. Muthitacharoen, A., Chen, B., Mazières, D.: A Low-bandwidth Network File System. In: Symposium on Operating Systems Principles. pp. 174–187 (2001)
21. Paulo, J., Pereira, J.: A Survey and Classification of Storage Deduplication Systems. ACM Computing Surveys 47(1), 11:1–11:30 (2014)
22. Quinlan, S., Dorward, S.: Venti: A New Approach to Archival Storage. In: USENIX Conference on File and Storage Technologies. pp. 1–13 (2002)
23. Rhea, S., Geels, D., Roscoe, T., Kubiatowicz, J.: Handling churn in a DHT. In: Proceedings of the USENIX Annual Technical Conference (2004)
24. Stoica, I., Morris, R., Liben-Nowell, D., Karger, D.R., Kaashoek, M.F., Dabek, F., Balakrishnan, H.: Chord: a scalable peer-to-peer lookup protocol for internet applications. Networking, IEEE/ACM Transactions on 11(1), 17–32 (2003)
25. Xia, W., Jiang, H., Feng, D., Hua, Y.: Silo: A Similarity-Locality based Near-Exact Deduplication Scheme with Low RAM Overhead and High Throughput. In: USENIX Annual Technical Conference. pp. 26–30 (2011)