Clustering Based Data Migration in Deduplicated Storage

Roei Kisous

Clustering Based Data Migration in Deduplicated Storage

Research Thesis

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science

Roei Kisous

Submitted to the Senate
of the Technion — Israel Institute of Technology
Tammuz 5782 Haifa July 2022

This research was carried out under the supervision of Dr. Gala Yadgar, in the Henry and Marilyn Taub Faculty of Computer Science.

Some results in this thesis have been published as articles by the author and research collaborators in conferences and journals during the course of the author's research period, the most up-to-date versions of which being:

Roei Kisous, Ariel Kolikant, Abhinav Duggal, Sarai Sheinvald, and Gala Yadgar. The *what*, the *from*, and the *to*: The migration games in deduplicated systems. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*, 2022.

Roei Kisous, Ariel Kolikant, Abhinav Duggal, Sarai Sheinvald, and Gala Yadgar. The *what*, the *from*, and the *to*: The migration games in deduplicated systems. Invited and submitted to the Special Section on FAST22 in the Transactions on Storage.

Acknowledgements

I would like to thank Dr. Gala Yadgar for putting in the time and effort that exceeded my expectations. The ideas you contributed, the brainstorming process that was used, the excellent guidance you provided, the inspiration you provided, and the pleasant process that you led made a significant difference in the outcome.

I would also like to thank Abhinav Duggal for his excellent advice and ideas, as well as providing an interesting viewpoint from the industry.

For their support and unconditional love, I am grateful to my parents, Meirav and Yosi in particular. Lastly, I would like to thank all of my friends for their support and friendship, which made the experience so much richer.

The generous financial help of the Technion is gratefully acknowledged.
This research was supported by the Israel Science Foundation (grant No. 807/20).

Contents

List of Figures

Abstract

Deduplication is a leading method for reducing physical storage capacity when duplicate data is present. This method can be applied on chunks, files, containers, and more. Instead of storing the same physical data multiple times, a pointer is created from each logical copy to the same physical copy, saving the space of the duplicate data. Due to this, data is shared between objects, such as files or entire directories, which result in garbage collection overhead and migration challenges.

In our work, we addressed the general migration problem where files are remapped between different volumes due to system expansion or maintenance. The question of which files and blocks to migrate has been extensively studied in systems without deduplication. However, only simplified migration problems have been considered in the context of deduplicated storage. As part of a migration plan, we aim to minimize the system's size while simultaneously ensuring that the storage load is evenly distributed across the volumes and that the network traffic required for the migration does not exceed its allocation.

Following that, we outline a way to develop effective migration plans using hierarchical clustering. Clustering refers to grouping objects based on their similarity. Hierarchical clustering, in particular, takes the distance between those objects into account. Each object is initially clustered separately, and the process of iterative clustering merges, in each step, two clusters with a minimal distance between them. We are interested in clustering files with high similarity together in order to reduce the amount of physical data while still maintaining low network traffic and a balanced system. Based on each cluster, we calculate data savings, traffic consumed, and load balance achieved and determine the plan's quality.

We show that this method has different tradeoffs between computation time and migration efficiency compared to other algorithms such as greedy and ILP (Integer Linear Programming). Our algorithm achieves almost identical results (and sometimes even better) than ILP, which, theoretically, is optimal, but in much less time.

Chapter 1

Introduction

Deduplication of data is often used in large-scale storage systems in order to reduce their size. As a result of deduplication, multiple files with duplicate data blocks are identified and replaced with pointers pointing to the same block in the system. Although the size of the system is reduced, there is an increase in its complexity. Academic studies and commercial systems have already addressed the complexity of reading, writing, and deleting data in deduplicated storage systems, but there are still high-level management aspects of large-scale systems to be addressed, such as capacity planning, caching, and quality and cost factors [SCJ16].

We have been investigating the process of *data migration*, which involves moving files between deduplication domains, or *volumes*. One volume can represent a single machine within a larger system, or it may represent a group of machines dedicated to a single customer or dataset. The system might remap files if a volume reaches its capacity limit or if another bottleneck forms. Due to the data dependencies between files introduced by deduplication, selecting which files to migrate requires new considerations: when a file is migrated, some of its blocks may be removed from the original volume, while others may still belong to other files in the volume. Similar to this, some blocks may need to be transferred to the target volume, while others might be already present. A successful migration plan aims to optimize several objectives, some of which may be in conflict: physical size of the data after migration, load balancing of the physical data stored among system volumes, and network bandwidth generated by the migration process itself.

Numerous studies have described simplified (case-specific) data migration in deduplicated systems in recent years. As part of their study, Harnik et al. [HHS$^+$19] discussed capacity estimation and present a greedy algorithm for reducing system size. Rangoli [NK13] deletes files as part of a greedy algorithm to reclaim some of the system's capacity. By using ILP (integer linear programming), GoSeed [NSKY21] solves the seeding problem of remapping files into an initially empty volume, called *seeding*.

Our work deals with the general case of data migration. We can think of the data migration problem in its most general form as an optimization problem that aims

to minimize the overall size of the system. The migration plan should also consider traffic and load balancing. Having these constraints enforced to a specific degree allows administrators to prioritize different costs by making good use of the solution space. Consequently, the challenge of data migration in deduplicated systems is to choose what to migrate, where to migrate from, and how to migrate with the constraints of traffic and load balancing specified by administrators. On the general case of data migration, an ILP approach and an extended greedy approach [KKD+22] of Harnik et al.'s [HHS+19] were applied.

Using hierarchical clustering, we introduce an algorithm that, to the best of our knowledge has never been applied to data deduplication. Files similar to each other are grouped into clusters, where the target number of clusters equals the number of volumes in the system. Clustering incorporates the physical location of the files, such that the similarity between files reflects their shared blocks and their initial locations. The migration plan assigns clusters to volumes based on the existing blocks on the volumes, and it remaps each file to its assigned cluster.

Our algorithm was implemented and evaluated on six system snapshots derived from three public datasets [FSL, MB11, Lin]. Using our algorithm, we demonstrate that we can reduce the system's size while complying with traffic and load balancing constraints. The clustering algorithm obtains results that are comparable, and sometimes even better, than the theoretically optimal ILP-based approach. Furthermore, the clustering algorithm runs much faster. It also performs better than the greedy algorithm.

Chapter 2

Background and related work

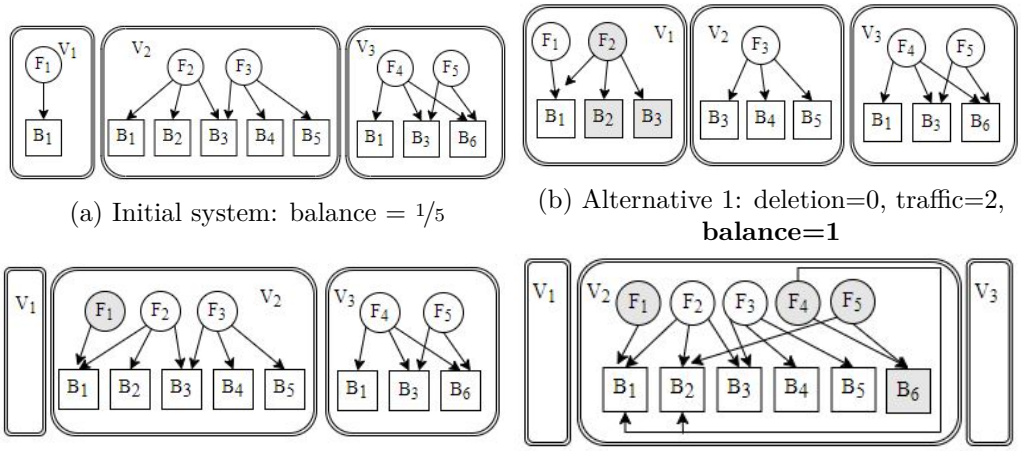
2.1 Data deduplication

By definition, deduplication means splitting incoming data into separate chunks known as *blocks*. Each block must be hashed to create a *fingerprint* that can be used to identify duplicate blocks and retrieve their unique copies from storage. This process requires the optimization of several aspects to maintain the storage system's performance. A few of them are chunking and fingerprinting [Man94, XJF+14, XZJ+16, MCM01, AAA+10], indexing and lookups [ZLP08, SBGV12, ADK+18], as well as reassembling files quickly [LEB+09, LSD+14, DSL10, SBGV12, CLZ11, YJTL16, LLD+14]. Though deduplication was initially used for backup and archiving purposes, it is now increasingly used for primary storage.

2.2 Data migration in distributed deduplication systems

Many distributed deduplication designs have been introduced in academic and commercial studies [CAVL09, DGH+09, GE11]. Here, we emphasize those that contain a separate fingerprint index for each physical server [DDL+11, BELL09, HHS+19, BLC14, DDS+17]. The design choice not only maintains a small index size and a low lookup cost, it also simplifies the client-side logic and facilitates garbage collection on the server side. This design defines a *deduplication domain* by a server (*volume*), which means, for instance, that duplicate blocks can only be found within that volume. Thus, files that are mapped to a specific volume refer to blocks that are physically present on that volume.

Traditional distributed systems differ from deduplicated systems in that striping files across volumes reduces deduplication even if using a content-based chunking algorithm. It also makes garbage collection more difficult when splitting files across clusters. Additionally, many storage systems (such as those from IBM [HHS+19] and DataDomain [DJS+19]), which operate as independent "islands" of storage within the data center or across data centers, perform deduplication within each individual subsystem,



(a) Initial system: balance = 1/5

(b) Alternative 1: deletion=0, traffic=2, **balance=1**

(c) Alternative 2: deletion=1/9, **traffic=0**, balance=0

(d) Alternative 3: **deletion=**3/9, traffic=1, balance=0

Figure 2.1: Initial system (a) and alternative migration plans: with optimal balance (b), optimal traffic (c), and optimal deletion (d). All the blocks in the system are of size 1.

as well as migrating files between them to re-balance the system as a whole.

If a subsystem becomes full but a different subsystem has capacity, migration may be a better and cheaper alternative than adding capacity to a full system. Existing mechanisms move only the metadata of files and chunks that are not already present in the target subsystem [DJS+19]. As with typical backup customers, monthly migration aligns with average retention periods.

Due to the relationship between the logical file's location and the physical location of its blocks, when a file is remapped from its volume, all its blocks must be relocated to the new volume. Furthermore, the file's blocks may belong to other files, so they cannot be necessarily removed from their original volume. In Figure 2.1(a), remapping file F_2 from volume V_2 to volume V_1 results in the system in Figure 2.1(b). Since Block B_1 is already present on V_1, it is deleted from V_2. Additionally, Block B_2 is not present on V_1 in the initial system, and Block B_3 is also absent, so both are moved to V_1. Instead of being deleted from V_2 as in the case of B_2, B_3 belongs to F_3 and must be replicated instead of being moved. Both the original system and this alternative have nine blocks as their total size.

2.3 Fingerprint sampling

Traditionally, sampling is used to manage large problems and has been used in deduplication systems to enhance the efficiency of the deduplication process [LEB+09, BELL09, CLZ11], route streams to servers [DDL+11], estimate deduplication ratios [HKS16], and manage volume capacity [HHS+19]. Using a sampling degree of k includes all chunks with k leading zeros in their fingerprints, as well as those files containing those chunks. The sample will contain $\frac{1}{2^k}$ of the original chunks if the fingerprint values are uniformly distributed, so the sample will simulate a much smaller system. Therefore, performing calculations on the sampled system should be much easier and faster. Harnik et

al. [HHS+19] demonstrate that $k = 13$ is sufficient to estimate the capacity of deduplicated volumes larger than 100GB with small errors. With sampling, the size of the problem instance is reduced by a predictable factor: increasing the sampling degree k by one results in a halving of the number of blocks.

2.4 Existing data migration approaches

The greedy iterative algorithm of Harnik et al. [HHS+19] aims to reduce the capacity in multi-volume deduplicated systems. Files are selected based on the space saving ratio from remapping a specific file to each of the other volumes: the amount of space it will take up in the target volume divided by the amount of space that can be reclaimed from the source volume. The mapping with the lowest ratio is remapped to a new volume at each iteration and the procedure is repeated until the desired deletion has been accomplished.

This approach led to the development of an iterative greedy algorithm for general migration [KKD+22]. The algorithm consists of a repeating phase with a predefined traffic allocation for each phase. To achieve the load balancing goal and minimize the capacity of the system, each phase consists of two alternating steps. The load balancing step focuses on load balancing the system. The goal is to remap a file between two volumes $\langle source, target \rangle$, with the source volume being the largest and the target volume being the smallest, for which such a file exists. During the capacity-reduction step, the goal is to reduce the overall capacity of the system without affecting load balance. As with the original greedy algorithm [HHS+19], the file with the lowest space saving ratio is selected for remapping while ensuring that no load balancing violation occurs.

GoSeed [NSKY21] addresses a simplified version of data migration known as *seeding*, in which all data is mapped to a single volume at the beginning. This migration aims to erase some of a volume's blocks by mapping files to the empty target volume [DJS+19]. The seeding problem is modeled as an ILP (integer linear programming) problem in GoSeed. GoSeed's solution determines which files and blocks are *moved* from the source volume to the target volume, and which are *replicated* to create copies on both volumes. The existence of open-source [SYM, lps, GNU] and commercial [CPL, Gur] ILP-solvers enables the efficient solution of this NP-hard problem using heuristics.

[KKD+22] addresses general migration by using ILP. In this approach, the main objective is to minimize the system's capacity, as in GoSeed, while both network traffic and load balancing constraints are enforced to ensure a valid migration plan.

Rangoli [NK13] is an algorithm for *space reclamation* that selects a set of files for deletion to reduce the physical size of the system. This is another specific case of data migration. Rangoli [NK13] groups files into roughly equal-sized bins based on the blocks they share, and then picks the bin with the smallest amount of data shared with other bins. Among the discussions in Shilane et al.'s [SCJ16] are additional data

migration scenarios and their resulting difficulties in deduplicated systems.

Chapter 3

Motivation and problem statement

3.1 Minimizing migration traffic

High-performance storage systems typically limit the portion of their network bandwidth that can be used for maintenance tasks such as reconstruction of data from failed storage nodes [RSG+13, HSX+12]. Data migration naturally involves significant network bandwidth consumption, and traditional data migration plans and mechanisms strive to minimize their network requirements as one of their optimization goals [LAW02, MHS18, TAB11, DJS+19, AHK+02, AHH+01]. In this work, we focus on the amount of data that is moved between nodes. The physical layout of the nodes and the precise scheduling of the migration are outside the scope of our current work.

In deduplicated storage, we distinguish between two costs associated with data migration. The *migration traffic* is the amount of data that is transferred between volumes during migration. The *replication cost* is the total size of duplicate blocks that are created as a result of remapping files to new volumes. Previous studies of data migration in deduplicated systems did not consider bandwidth explicitly. Harnik et al. [HHS+19] did not address this aspect at all. In the seeding problem addressed by GoSeed [NSKY21], the migration traffic is implicitly minimized as a result of minimizing the replication cost. In the general case, however, migration traffic is potentially independent of the amount of data replication.

For example, Alternative 1 in Figure 2.1(b) results in transferring two blocks, B_2 and B_3, between volumes, even though B_2 is eventually deleted from its source volume. In contrast, the alternative migration plan in Figure 2.1(c) does not consume traffic at all: file F_1 is remapped to V_2 which already stores its only block, and thus B_1 can simply be deleted from V_1. This alternative also reduces the system's size to eight blocks, making it superior to the first alternative in terms of both objectives. We note, however, that this is not always the case, and that minimizing the overall system size

and minimizing the amount of data transferred might be conflicting objectives.

3.2 Load Balancing

Load balancing is a major objective in distributed storage systems, where it often conflicts with other objectives such as utilization and management overhead [AHK+02, WBMM06, NEF+12]. Distributed deduplication introduces an inherent tradeoff between minimizing the total physical data size and maximizing load balancing: the system's size is minimized when all the files are mapped to a single volume, which clearly gives the worst possible load balancing. Thus, distributed deduplication systems weigh the benefit of mapping a file to the volume that contains similar files, against the need to distribute the load evenly between the volumes. Load balancing can refer to various measures of load, such as IOPS, bandwidth requirements, or the number of files mapped to each volume.

We follow previous work and aim to evenly distribute the *capacity load* between volumes [BELL09, DDL+11, KKD+22]. Balancing capacity is especially important in deduplicated systems that route incoming files to volumes that already contain similar files. In such designs, volumes whose storage occupancy is slightly higher than others might quickly become overloaded due to their larger amount of data 'attracting' even more new files, and so on. Capacity load balancing can be expected to lead to better network utilization and prevent specific volumes from becoming a bottleneck or exhausting their capacity. While performance load balancing is not our main objective, we expect it to improve as a result of distributing capacity.

In this work, we capture the load balancing in the system with the *balance* metric, which is similar to a commonly used *fairness* metric [GWM07]—the ratio between the size of the smallest volume in the system and that of the largest volume. For example, the balance of the initial system in Figure 2.1(a) is $|V_1|/|V_2| = 1/5$. Alternative 1 (Figure 2.1(b)) is perfectly balanced, with $balance = 1$, while Alternative 2 (Figure 2.1(c)) has the worst balance: $|V_1|/|V_2| = 0$.

3.3 Problem statement

There are various approaches for handling conflicting objectives in complex optimization systems. These include searching for the Pareto frontier [ZKT08], or defining a composite objective function of weighted individual objectives. We chose to keep the system's size as our main objective, and to address the migration traffic and load balancing as constraints on the migration plan. We define the general migration problem by extending the seeding problem from [NSKY21], and thus we reuse some of their notations for compatibility.

For a storage system S with a set of volumes V, let $B = \{b_0, b_1, \ldots\}$ be the set of unique blocks stored in the system, and let $size(b)$ be the size of block b. Let

$F = \{f_0, f_1, \ldots\}$ be the set of files in S, and let $I_S \subseteq B \times F \times V$ be an inclusion relation, where $(b, f, v) \in I_S$ means that file f mapped to volume v contains block b which stored in this volume. We use $b \in v$ to denote that $(b, f, v) \in I_S$ for some file f. The size of a volume is the total size of the blocks stored in it, i.e., $size(v) = \Sigma_{b \in v} size(b)$. The size of the system is the total size of its volumes, i.e., $size(S) = size(V) = \Sigma_{v \in V} size(v)$.

The *general migration problem* is to find a set of files $F_M \subseteq F$ to migrate, the volume each file is migrated to, the blocks that can be deleted from each volume, and the blocks that should be copied to each volume. Applying the migration plan results in a new system, S'. The *migration goal* is to minimize the size of S'. This is equivalent to maximizing the size of all the blocks that can be deleted during the migration, minus the size of all the blocks that must be replicated.

The *traffic constraint* specifies T_{max}—the maximum traffic allowed during migration. It requires that the total size of blocks that are added to volumes they were not stored in is no larger than T_{max}. The *load balancing constraint* determines how evenly the capacity is distributed between the volumes. It specifies a *margin* $0 \leq \mu < 1$ and requires that the size of each volume in the new system is within μ of the average volume size. For a system with $|V|$ volumes, this is equivalent to requiring that $balance \leq {}^{[size(S')/|V| \times (1-\mu)]}/_{[size(S')/|V| \times (1+\mu)]}$.

For example, for the initial system in Figure 2.1(a), Alternative 1 (Figure 2.1(b)) is the only migration plan that satisfies the load balancing constraint (for any μ). For T_{max} lower than $2/9$, no migration is feasible. On the other hand, if we remove the load balancing constraint, the optimal migration plan depends on the traffic constraint alone: Alternative 2 (Figure 2.1(c)) is optimal for, e.g., $T_{max} = 0$, and Alternative 3 (Figure 2.1(d)) is optimal for $T_{max} = 3$.

3.4 Refinements

This generalized problem can be refined in several straightforward ways. For example, we can restrict the set of files that may be included in F_M, the set of volumes from which files may be removed, or the set of volumes to which files can be remapped. Similarly, we can require that a specific volume be removed from the system (enforcing all its files to be remapped), or that an empty volume be added. We demonstrate some of these cases in our evaluation.

Chapter 4

Clustering

4.1 Overview

Clustering is a well-known technique for grouping objects based on their similarity [clu]. It is fast and effective, and is directly applicable to our domain: files are similar if they share a large portion of their blocks. Our approach is thus to create clusters of similar files and to assign each cluster to a volume, remapping those files that were assigned to a volume different from their original location. Despite its simplicity, three main challenges ($Ch1 - Ch3$) are involved in applying this idea to the general migration problem.

($Ch1$) **Unpredictable traffic** The traffic required for a migration plan can only be calculated after the clusters have been assigned to volumes. When the clustering decisions are being made, their implications on the overall traffic are unknown and thus cannot be taken into consideration.

($Ch2$) **Unpredictable system size** The load-balancing constraint is given in terms of the system's size after migration. However, this size is required to ensure, during the clustering process, that the created clusters are within the allowed sizes.

($Ch3$) **High sensitivity** The file similarity metric is based on the precise set of blocks in each file. When this metric is applied to a sample (subsection 2.3) of the storage system's fingerprints, it is highly sensitive to the sampling degree and rule.

We address these challenges with several heuristics ($H1 - H4$):

($H1$) **Traffic weight** We define a new similarity metric for files. This metric is a weighted sum of the files' content similarity and a new distance metric that indicates how many source volumes contain files within a cluster. Our algorithm considers files as similar if they contain the same blocks and are mapped to the same source volume. Assigning a higher weight (W_T) to the content similarity will result in a smaller system but higher migration traffic.

($H2a$) **Estimated system size** We further use the weight to estimate the size of the system after migration. We calculate the size of a hypothetical system without du-

plicates, and predict that higher migration traffic will bring the system closer to this hypothetical optimum.

($H2b$) **Clustering retries** We use the estimated final system size to determine the maximum allowed cluster size. During the clustering process, we stop adding files to clusters that reach this size. If the process halts due to this limitation, we increase the maximum size by a small margin, and restart it.

($H3$) **Randomization** Instead of deterministic clustering decisions, we choose a random option from the set of best options. Different random seeds potentially result in different systems.

($H4$) **Multiple executions** Our heuristics introduce several parameters which we would be loath to overfit. We use the same initial state to perform repeated executions of the clustering process with multiple sets of parameter combinations (180 in our case), and choose the best migration plan from those executions as our final output.

In the following, we give the required background on the clustering process and describe each of our optimizations in detail.

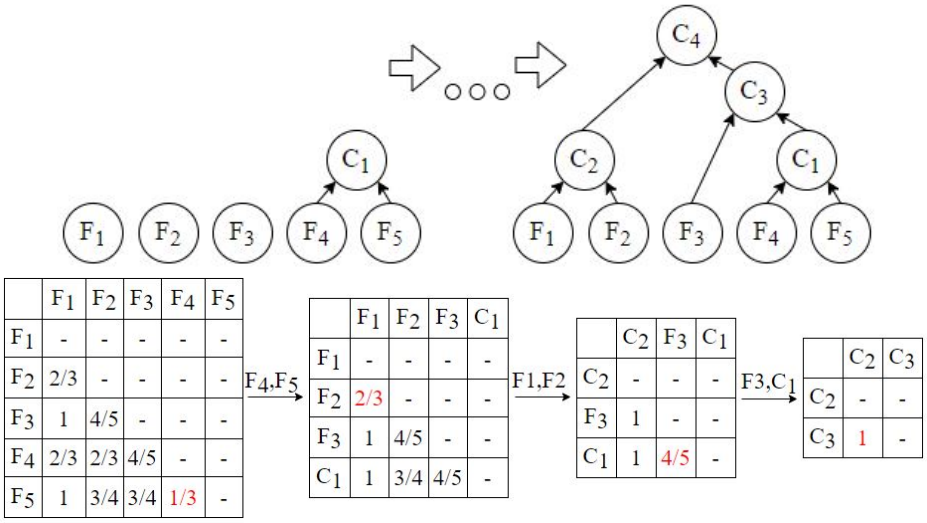


Figure 4.1: Hierarchical clustering with the files from Figure 2.1 (top) and the distance matrices created in the process (bottom).

4.2 Hierarchical clustering

Hierarchical clustering [GP13] is an iterative clustering process that, in each iteration, merges the most similar pair of clusters into a new cluster. The input is an initial set of objects, which are viewed as clusters of size 1. The process creates a tree whose leaves are the initial objects, and internal nodes are the clusters they are merged into. For example, Figure 4.1 shows the clustering hierarchy created from the set of initial objects $\{F_1, ..., F_5\}$, where the clusters $\{C_1, ..., C_4\}$ were created in order of their indices.

Hierarchical clustering naturally lends itself to grouping of files. Intuitively, files that share a large portion of their blocks are similar and should thus belong to the same cluster and eventually to the same volume. For example, the initial objects in Figure 4.1 represent the files in Figure 2.1(a): F_4 and F_5 share two blocks and are thus merged into the first cluster, C_1. Our clustering-based approach is simple: we group the files into a number of clusters equal to the number of volumes in the system and assign one cluster to each volume. This assignment implies which files should be remapped and which blocks should be transferred and/or deleted in the migration. For example, for a system with three volumes, we could halt the clustering process in Figure 4.1, resulting in a final set of three clusters: $\{C_1, C_2, F_3\}$. We develop this basic approach to the general migration problem, i.e., to maximize the deletion and to comply with the traffic and load-balancing constraints.

4.3 File similarity

The hierarchical clustering process relies on a similarity function that indicates which pair of clusters to merge in each iteration. We use the commonly used *Jaccard index* [GP13] for this purpose. For two sets A and B, their Jaccard index is defined as $J(A,B) = {}^{|A \cap B|}/_{|A \cup B|}$. We view each file as a set of blocks, and thus, the Jaccard index for a pair of files is the portion of their blocks that are shared. From hereon, we refer to the complement of the index: the *Jaccard distance* which is defined as $dist_J = \overline{J(A,B)} = 1 - J(A,B)$. This is to comply with the standard terminology in which the two clusters with the smallest distance are merged in each iteration. For example, the leftmost table in Figure 4.1 depicts the *distance matrix* for the files in Figure 2.1. Indeed, the distance is smallest for the pair F_4 and F_5 which are the first to be merged.

The Jaccard distance could easily be applied to entire clusters, which can themselves be viewed as sets of blocks. However, calculating the distance between each new cluster and all existing clusters would require repeated traversals of the original file blocks in each iteration. This complexity is addressed in hierarchical clustering by defining a *linkage* function, which determines the distance between the merged cluster and existing clusters based on the distances before the merge. Let A,B,C each be a cluster, where A and B are merged into a single cluster. We use complete linkage to define the distance between the newly merged cluster and the remaining clusters in the system: $dist_J(A \cup B, C) = max\{dist_J(A,C), dist_J(B,C)\}$. For example, the row for C_1 in the second distance matrix in Figure 4.1 lists the distances between C_1 and each of the remaining files.

4.4 Traffic considerations ($H1$)

We limit the traffic required by our migration plan in two ways. The first is by assigning each of the final clusters to the volume that contains the largest number of its blocks. We calculate the size of the intersection (in terms of the size of the shared blocks) between each cluster and each volume in the initial system. We then iterativly pick the $\langle cluster, volume \rangle$ pair with the largest intersection from the clusters and volumes that have not yet been assigned.

This assignment alone might still result in excessive traffic, especially if highly similar files are initially scattered across many different volumes. To avoid such situations, we incorporate the traffic considerations into the clustering process itself. Namely, we define the *volume distance*, $dist_V(C)$, of a cluster as the portion of the system's volumes whose files are included in the cluster. For example, in Figure 4.1, $dist_V(C_1) = 1/3$ and $dist_V(C_2) = 2/3$.

We then define a new *weighted distance* metric that combines the Jaccard distance and the volume distance: $dist_W(A, B) = W_T \times dist_J(A, B) + (1 - W_T) \times dist_V(A \cup B)$, where $0 \leq W_T \leq 1$ is the *traffic weight*. Intuitively, increasing W_T increases the amount of traffic allocated for the migration, which increases the priority of deduplication efficiency over the network transfer cost. Nevertheless, it does not guarantee compliance with a specific traffic constraint as shown in subsection 6.2.2. We address this limitation by multiple executions, described below.

4.5 Load-balancing considerations ($H2$)

We enforce the load balancing constraint by preventing merges that result in clusters that exceed the maximal volume size. We determine the maximal cluster size by estimating the system's size *after* migration. Intuitively, we expect that increasing the traffic allocated for migration will increase the reduction in system size, and we estimate this traffic with the W_T weight described above. Formally, we estimate the size of the final system as $Size(W_T) = W_T \times Size_{uniq} + (1 - W_T) \times size(S_{init})$, where $Size_{uniq}$ is the size of all the unique blocks in the system. The maximal cluster size is thus $C_{max} = {}^{Size(W_T)}/_{|V|}$

In each clustering iteration, we ensure that the merged cluster is not larger than C_{max}. This requirement might result in the algorithm halting before the target number of clusters is reached, due to merging decisions made earlier in the process. If this happens, we increase the value of C_{max} by a small ϵ and retry the clustering process, although it might also not adhere to our original load balancing constraint. We continue retrying until the algorithm creates the required number of clusters. A small ϵ can potentially yield the most balanced system, but might require excessively many retries. We use $\epsilon = 5\%$ as our default.

For instance, using the system in figure 2.1(a) with $W_T = 0.1$, we can deduct

C_{max}. First, we calculate that $Size_{uniq} = 6$, since there are 6 blocks in the system. Then, we infer that $size(S_{init}) = 9$, because there are 9 physical blocks scattered across the initial system. From that, it is easy to calculate $Size(W_T)$ using our $W_T = 0.1$: $Size(W_T) = 0.1 \times 6 + (1 - 0.1) \times 9 = 8.7$ and C_{max} which is $Size(W_T)/|V| = 8.7/3 = 2.9$. Any pair of clusters merged together would yield a cluster larger than C_{max} in size because any pair would contain at least 3 unique blocks. Therefore we increase C_{max} by $\epsilon = 5\%$ to $C_{max} = 3.045$ and repeat the clustering process successfully with the merge of F_4 and F_5 of size 3, which was previously forbidden.

4.6 Sensitivity to sample $(H3)$

We apply the hierarchical clustering process to a sample of the system, rather than to the complete set of blocks which can be excessively large. However, it turns out that the Jaccard distance is highly sensitive to the precise set of blocks that represent each file in the sample. We found, in our initial experiments, that different sampling degrees as well as different sampling rules (e.g., k leading ones instead of k leading zeroes in the fingerprint) result in small differences in the Jaccard distance of the file pairs, as described in subsection 6.2.2.

Such small differences might change the entire clustering hierarchy, even if the practical difference between the pairs of files is very small. Thus, rather than merging the pair of clusters with the smallest distance, we merge a *random* pair from the set of pairs with the smallest distances. We include in this set only pairs whose distance is within a certain percentage of the minimum distance. Thus, for a maximum distance difference *gap*, we choose a random pair$\langle C_i, C_j \rangle$ from the 10 (or less) pairs for which $Dist_W(C_i, C_j) \leq$ minimum distance $\times (1 + gap)$.

4.7 Constructing the final migration plan $(H4)$

The main advantage of our clustering-based approach is its relatively fast runtime. Constructing the initial distance matrix for the individual files is time consuming, but the same initial matrix can be reused for all the consecutive clustering processes on the same initial system. We exploit this advantage to eliminate the sensitivity of our clustering process to the many parameters introduced in this chapter. For the same given system and migration constraints, we execute the clustering process with six traffic weights ($W_T \in \{0, 0.2, 0.4, 0.6, 0.8, 1\}$), three gaps ($gap \in \{0.5\%, 1\%, 3\%\}$), and ten random seeds. This results in a total of 180 executions, some of which are performed in parallel (depending on the resources of the evaluation platform). We calculate the deletion, traffic, and balance of each migration plan (on the sample used as the input for clustering), and as our final result, use the plan with the best deletion that satisfies the load-balancing and traffic constraints.

We also include in our evaluation a relaxed scheme without the load-balancing

constraint (i.e., $C_{max} = \infty$). In this scheme, the final migration plan must only satisfy the traffic constraint.

Chapter 5

Implementation

The clustering algorithm creates a $|F| \times |B|$ bit matrix to indicate whether each file contains each block, and uses it to create the distance matrix (see Figure 4.1). The clustering process uses and updates only the lower triangular of this matrix. We use the upper triangular to record the initial distances, and to reset the lower triangular when the clustering process is repeated for the same system and different input parameters (W_T, *gap*, or random seed). When the clustering process completes, we use the file-block bit matrix to determine the assignment of clusters to volume. Our program consists of approximately 1000 lines of `C++` code and is available online [Git]. Each clustering process is performed on a private copy of the distance matrix within a single thread, and our evaluation platform is sufficient for executing six processes in parallel.

Chapter 6

Evaluation

6.1 Experimental setup

We ran our experiments on a server running Ubuntu 18.04.3, equipped with 128GB DDR4 RAM (with 2666 MHz bus speed), Intel® Xeon® Silver 4114 CPU (with hyper-threading functionality) running at 2.20GHz, one Dell®T1WH8 240GB TLC SATA SSD, and one Micron 5200 Series 960GB 3D TLC NAND Flash SSD.

File system snapshots. We used two static file system snapshots from datasets used to evaluate the seeding problem [NSKY21]: The UBC dataset [MB11, SNI] includes file systems of 857 Microsoft employees, of which we used the first 500 file systems (UBC-500). The FSL dataset [FSL] consists of snapshots of students' home directories at the FSL Lab at Stony Brook University [SKM+16, TMB+12]. We used nine weekly snapshots of nine users between August 28 and October 23, 2014 (Homes). These snapshots include, for each file, the fingerprints of its chunks and their sizes. Each snapshot file represents one entire file system, which is the migration unit in our model, and is represented as one file in our migration problem instances.

We created two additional sets of snapshots from the Linux version archive [Lin] obtained from [Ded22]. Our Linux-all dataset includes snapshots of all the versions from 2.0 to 5.9.14. We also created a smaller dataset, Linux-skip, which consists of every 5th snapshot. The latter dataset is logically (approximately) 5× smaller than the former, although their physical size is almost the same.

The UBC-500 and Homes snapshots were created with variable-sized chunks with Rabin fingerprints, whose specified average chunk size is 64KB. We created the Linux snapshots with an average chunk size of 8KB, because they are much smaller to begin with. We used these sets of snapshots to create six initial systems, with varying numbers of volumes. They are listed in Table 6.1. We emulate the ingestion of each snapshot into a simplified deduplication system which detects duplicates only within the same volume. In the UBC and Linux systems we assigned the same number of arbitrary snapshots to each volume. In the Homes-week system, we assigned snapshots from the same week to the same volume, such that each volume contains all the users' snapshots

System	Files	$\|V\|$	Chunks	Dedupe	Logical
UBC-500	500	5	382M	0.39	19.5 TB
Homes-week	81	3	19M	0.38	8.9 TB
Homes-user	81	3	19M	0.16	8.9 TB
Linux-skip	662	5 / 10	1.76M	0.12 / 0.19	377 GB
Linux-all	2703	5	1.78M	0.03	1.8 TB

Table 6.1: System snapshots in our evaluation. $|V|$ is the number of volumes, Chunks is the number of unique chunks, and Dedupe is the deduplication ratio—the ratio between the physical and logical size of each system. Logical is the logical size.

from a set of three weeks. In the Homes-user system, we assign each user to a dedicated volume such that each volume contains all the weekly snapshots of a set of three users.

The algorithm is executed on a sample of the system's fingerprints, to reduce its memory consumption and runtime. We use a sampling degree of $k = 13$ unless stated otherwise. The final system size after migration, as well as the resulting balance and consumed traffic are calculated on the original system's snapshot. We use a calculator similar to the one used in [NSKY21]: we traverse the initial system's volumes and sum the sizes of blocks that remain in each volume after migration and those that are added to the volume as a result of it. We experimented with three T_{max} values, 20%, 40%, and 100% of each system's initial size, and three μ values, 2%, 5%, and 10% of the system size after migration.

6.2 Experimental results

We wish to answer two main questions: (1) how does the clustering algorithm (*Cluster*) compare in terms of the final system size, load balancing, and runtime to the other algorithms? and (2) how is the performance of the algorithm affected by the various system and problem parameters?

6.2.1 Basic comparison between algorithms

Figure 6.1 shows the *deletion*—percentage of the initial system's physical size that was deleted by each algorithm. The deletion is higher for systems that were initially more balanced, i.e., the Linux and Homes-weeks systems. For all the systems except UBC-500, Greedy achieved significantly smaller deletion than Cluster. In UBC-500, there is less similarity and therefore less dependency between files, which eliminates some of the advantage that Cluster and ILP have over Greedy, which outperforms them when $T_{max} = 100\%$.

ILP and Cluster achieve comparable deletions to one another, even though the ILP solver attempts to find the theoretically optimal migration plan. We distinguish between two cases when explaining this similarity. In the first case (Linux-skip and Homes), the ILP-solver produces an optimal solution on the system's sample, but it is not optimal when applied to the full (unsampled) system. The deletion of Cluster is

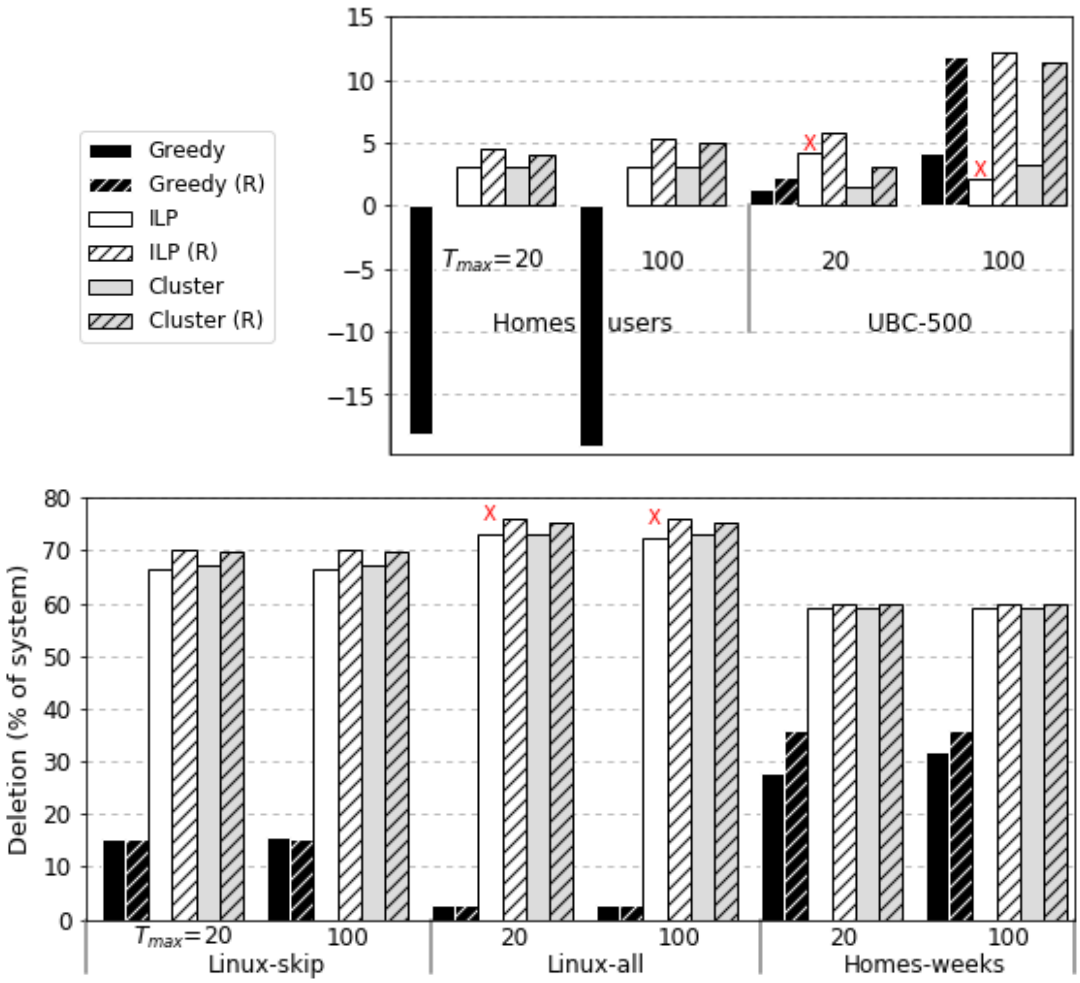


Figure 6.1: Reduction in system size of all systems and all algorithms (with and without load balancing constraints. $k = 13$ and $\mu = 2\%$).

up to 1% higher than that of ILP in those cases. In the second case, marked by a red 'x' in the figures, ILP times out (after six hours) and returns a suboptimal solution. Therefore, Cluster reaches a better solution.

The 'relaxed' (R) version of the algorithms, without the load balancing constraint, usually achieves a higher deletion than their full version. The largest difference is 558%, although the difference is typically smaller, and can be as low as 1.3%. In the case of Greedy in the Homes-users system, the relaxed version does not identify any file that can be remapped, and does not return any solution.

Figure 6.2 shows the balance achieved by each algorithm. With a margin of $\mu = 2\%$ and five volumes, the balance should be at least $18/22 = 0.82$. In practice, however, the balance might be lower, for two main reasons. Greedy might fail to bring the system to a balanced state if it exhausts (or thinks it exhausts) the maximum traffic allowed for migration. In contrast, Cluster and ILP generate a migration plan that complies with the load balancing constraint on the sample, but violates it when applied to the full (unsampled) system. The violation is highest in the Linux systems, where some files

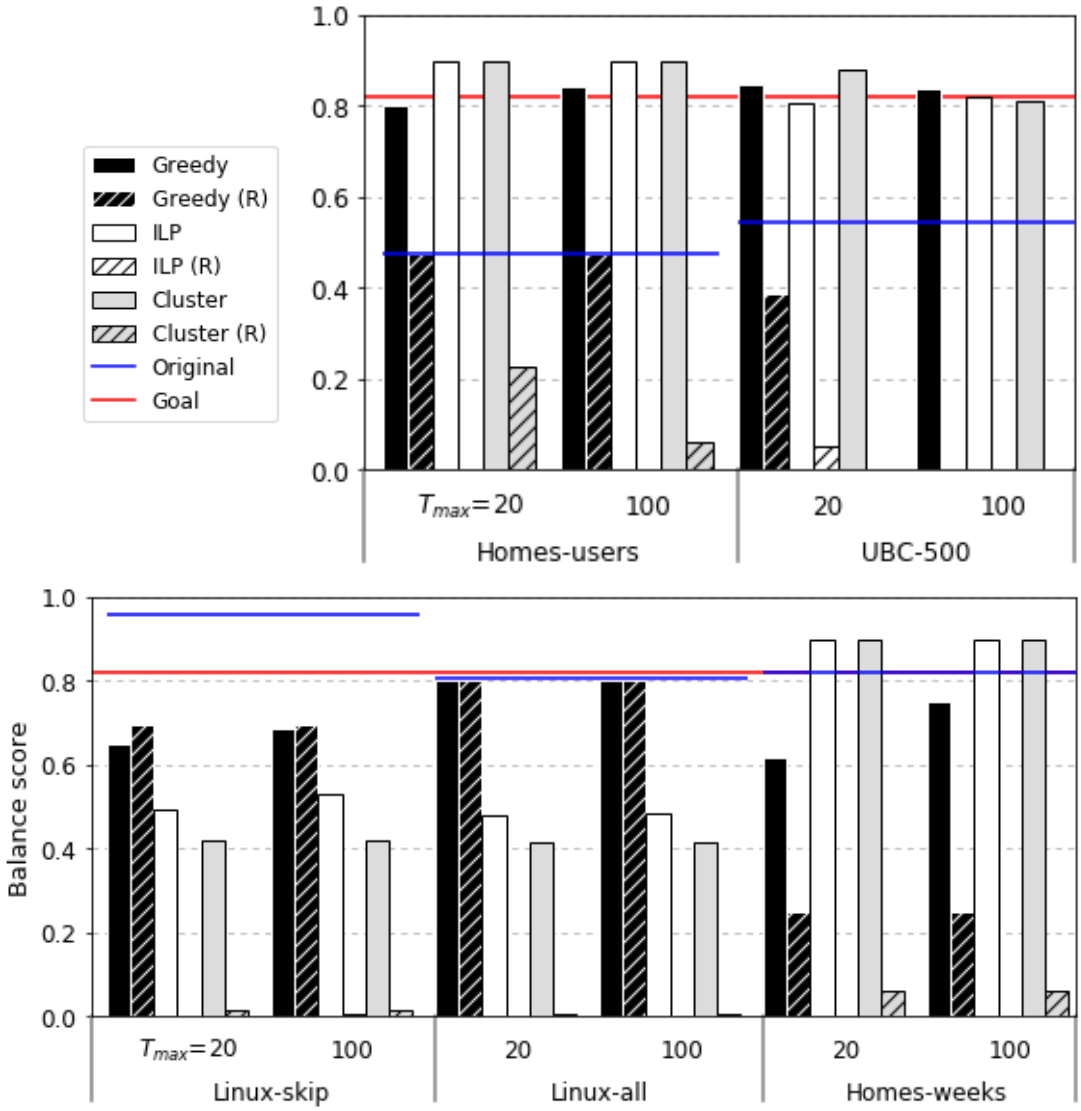


Figure 6.2: Resulting balance of all systems and all algorithms (with and without load balancing constraints. $k = 13$ and $\mu = 2\%$).

(i.e., entire Linux versions) consist of only one or two blocks. Nevertheless, specifying the load balancing constraint successfully improves the load balancing of the system. Without it, the relaxed Cluster and ILP versions create highly unbalanced systems, with some volumes storing no files at all, or very few small files. Greedy typically avoids such extremes, because it is unable to identify and group similar files in the same volume.

Figure 6.3 shows the runtime of each of the algorithms (note the log scale of the y-axis). Greedy generates a migration plan in the shortest runtime: 20 seconds or less. ILP requires the longest time, because it attempts to solve an NP-hard problem. Indeed, except for the Homes systems that have the fewest files, ILP requires more than an hour, and often halts at the six-hour timeout. The runtime of Cluster is longer than that of Greedy, and usually shorter than that of ILP. It is still relatively long, as

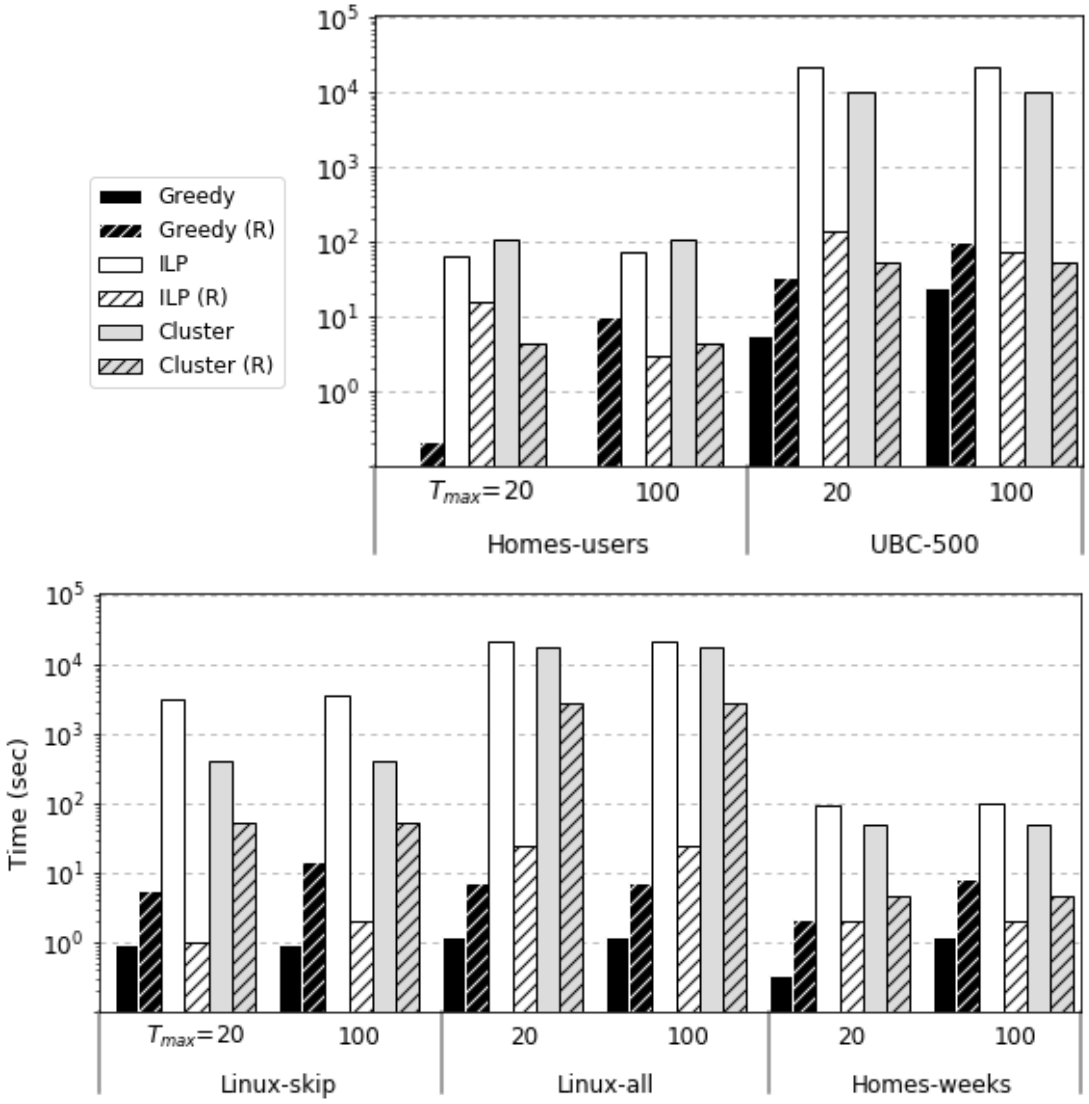


Figure 6.3: Algorithm runtime for all systems and all algorithms (with and without load balancing constraints. $k = 13$ and $\mu = 2\%$).

a result of performing 180 executions of the clustering process. We note, however, that this runtime can be shortened by reducing the number of executions, e.g., by reducing the number of random seeds or gaps. We evaluate the effect of these parameters in the following subsection.

Removing the load balancing constraint reduces the runtime of ILP and Cluster by one or two orders of magnitude. In ILP, this happens because the problem complexity is significantly reduced. In Cluster, the clustering is completed in a single attempt, without having to restart it due to illegal cluster sizes. Surprisingly, removing this constraint from Greedy increases its run time. The reason is that each iteration in the capacity-reduction step is much longer than those in the load-balancing step, as it examines all possible file remaps between all volume pairs in the system. In the relaxed Greedy version, all the traffic is allocated to capacity savings and thus its runtime

increases.

Implications. Our basic comparison leads to several notable observations. (1) Cluster and ILP have a clear advantage over Greedy. This was not the case in previous studies that examined simple cases of migration, i.e., seeding [NSKY21] and space reclamation [NK13]. However, the increased complexity of the general migration problem increases the gap between the greedy and the optimal solutions. (2) Cluster is comparable and might even outperform ILP, despite the premise of optimality of the ILP-based approach. This is a combination of the high complexity of the ILP problem with the ability to execute multiple clustering processes quickly and in parallel. We conclude that hierarchical clustering is highly efficient for grouping similar files, and that our heuristics for addressing the traffic and load balancing constraints are highly effective. (3) In most systems, adding the load balancing constraint limits the potential capacity reduction, but this limit is usually modest, i.e., several percents of the system's size. The extent of this limitation depends on the degree of similarity between files and the balance of the initial system.

6.2.2 Sensitivity to Cluster's configuration

Introduction. The fingerprint sampling described in subsection 2.3 refers to the method of selecting a portion of a dataset's blocks in a way that retains the unique characteristics of the dataset. We refer to the method by which we choose fingerprints as the *sampling rule*. *Sampling degree*, denoted by k, is the extent to which this rule is applied. If, for instance, the sampling degree is equal to 13 and the *leading-zeros* sampling rule is used, only the blocks whose bit-wise representation of the fingerprint has 13 leading zeros will be selected. Throughout our research, we found out that our algorithm is highly sensitive to both *sampling rule* and *sampling degree*. This section explores those sensitivities.

Effect of sampling rules. Sampling rule refers to the method by which we choose fingerprints. There are a few sampling rules we examined throughout our research. (1) *leading-zeros* - chooses only the blocks whose fingerprint contains k (sampling degree) leading zeros. (2) *leading-ones* - chooses only the blocks whose fingerprint contains k leading ones. (3) *alternating zero-one* - chooses only the blocks whose fingerprint contains k alternating zeros and ones. For this sampling rule, if $k = 7$, for example, only blocks whose fingerprint starts with 0101010 will be included in the sample.

Figure 6.4 shows ten pairs of files with their dissimilarities under those different sampling rules. The results suggest that different sampling methods result in varying dissimilarity values, as well as varying order of pairs based on that value, which dictates which files should be merged. Essentially, the algorithm does not consider the absolute dissimilarity values, but only the order of the pairs, as it picks the smallest pair to merge regardless of the absolute value. As an example, let's look at the pairs $\langle 0, 10\rangle$ (white), $\langle 0, 40\rangle$ (green) and $\langle 10, 40\rangle$ (black). In accordance with the *leading-*

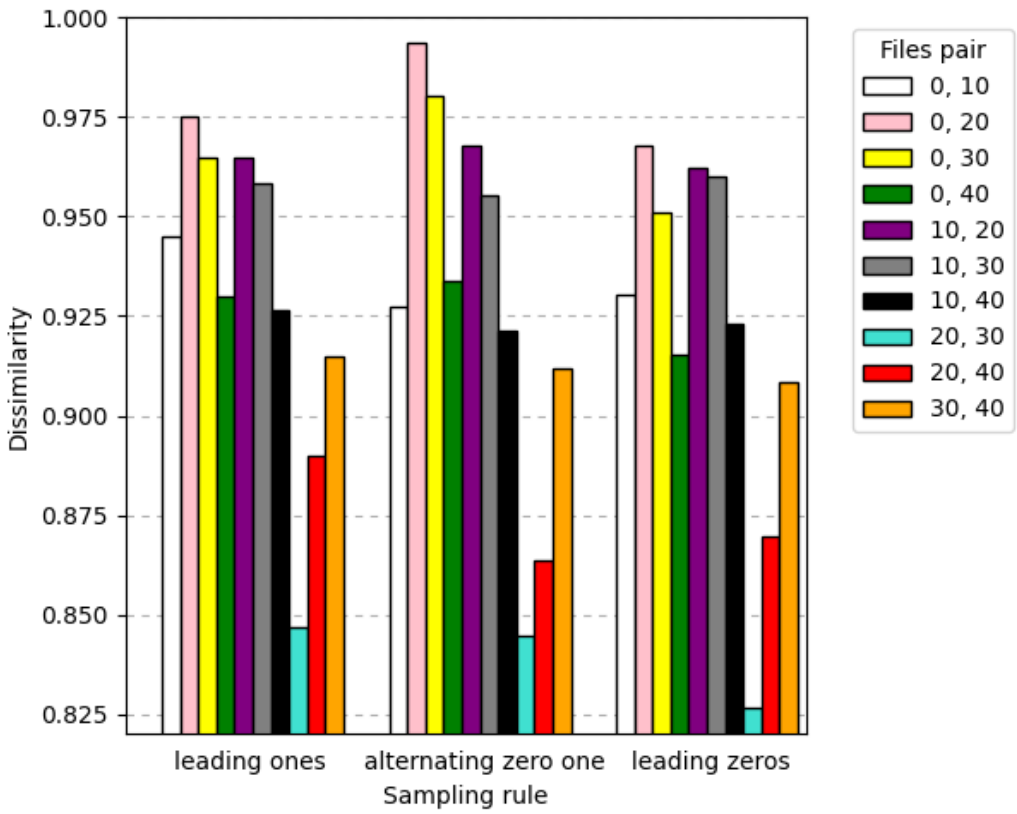


Figure 6.4: Dissimilarities under different sampling rules with $k = 13$.

zeros sampling rule, the pairs are ranked (best to worst, low to high) according to their dissimilarity value: $\langle 0, 40\rangle$, $\langle 10, 40\rangle$, $\langle 0, 10\rangle$. Alternatively, under *leading-ones*, the order becomes $\langle 10, 40\rangle$, $\langle 0, 10\rangle$, $\langle 0, 40\rangle$, while under *alternating zero-one*, the order becomes $\langle 10, 40\rangle$, $\langle 0, 40\rangle$, $\langle 0, 10\rangle$ which is also different. As we have found throughout our research, different merging decisions in the initial stages of the clustering process could result in a completely different migration plan. This observation motivated us to add randomization to the clustering process, as described in Section 6.2.2

Effect of randomization on Cluster. Figure 6.5 shows the range of deletion values and traffic usage of the migration plans generated by Cluster for Linux-all with $k = 13$. Each bar shows the 25th, 50th, and 75th percentiles, and the whiskers show the minimum and maximum values achieved with different random seeds for each combination of *gap* and W_T.

Our results show that different random seeds can result in large differences in the deletion and traffic: up to 84% and 400%, respectively, when W_T and *gap* are fixed. At the same time, W_T cannot predict the actual traffic used by the migration plan since it is only used heuristically to simulate the traffic constraint. This is the reason for repeating the clustering process for a range of W_T values. Indeed, different W_T values result in very different values of deletion. For a given W_T, the range of deletion and traffic values generated by different *gaps* are similar. Thus, as no *gap* consistently

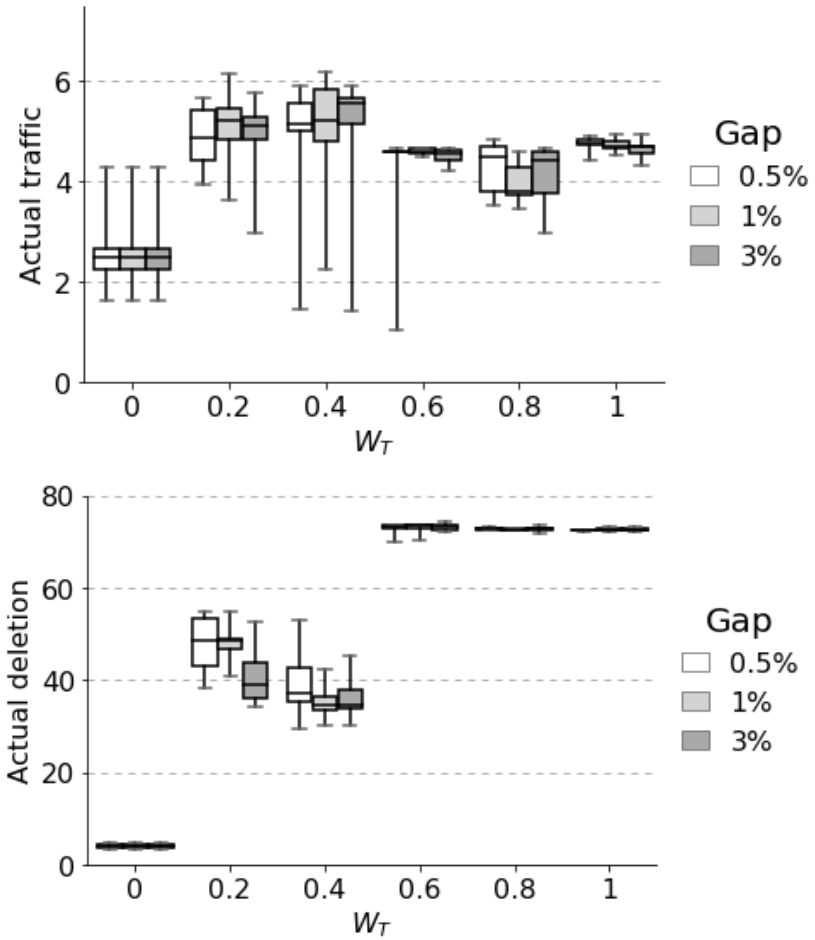


Figure 6.5: The distribution of migration traffic (top) and reduction in system size (bottom) in the set of plans returned by Cluster for Linux-all with $k = 13$.

outperforms the others, executing the clustering with one or two gaps instead of three will likely have a limited effect on the results while significantly reducing the runtime.

We repeated the same experiment with a different numbers of random seeds. Figure 6.6 shows that increasing the number of seeds from 5 to 15 (respectively increasing the number of runs from 90 to 270) carries diminishing returns. Thus, in practice, it is possible to halt the algorithm when additional runs do not improve the best solution so far.

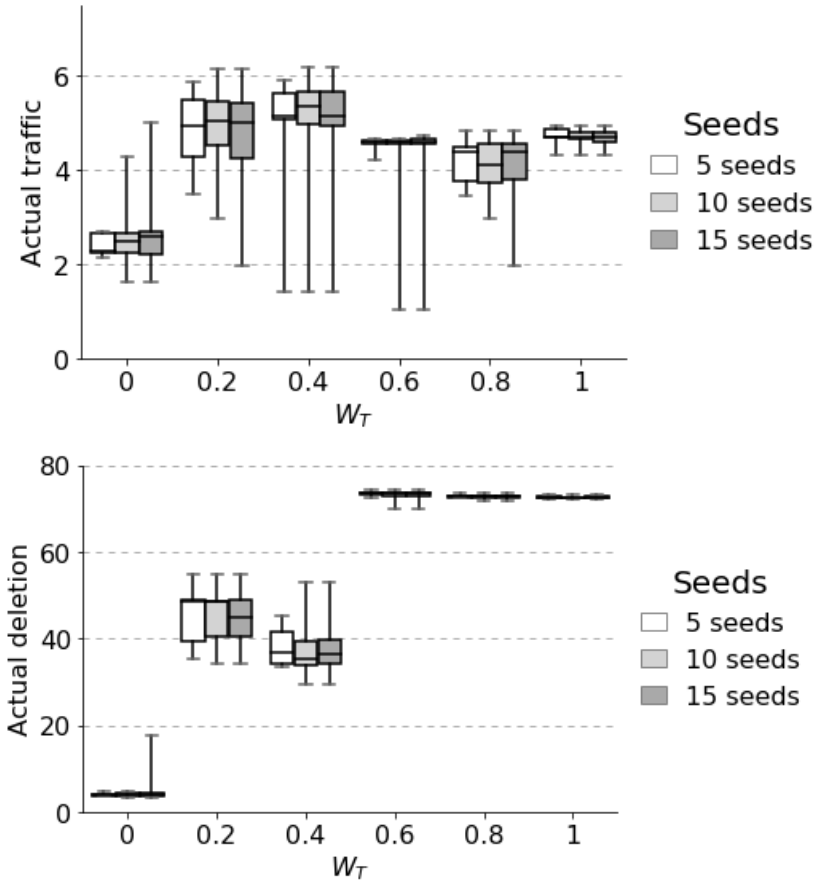


Figure 6.6: The distribution of migration traffic (top) and reduction in system size (bottom) in the set of plans returned by Cluster for Linux-all with $k = 13$ under different number of random seeds.

Finally, we examine whether the sensitivity to the configuration parameters differs between different systems. We execute Cluster with three gaps and 10 random seeds on three different systems. Figure 6.7 shows the results for each system and each value of W_T. As we expected, the traffic consumption is different in different systems. The results for homes-users demonstrate that smaller systems are more sensitive to this parameter, because each file consists of a larger portion of the entire system. Thus, when the traffic weight of the similarity metric is small (small W_T), there are limited options for clustering in the early stages of the process, and all random seeds result in the same plan. However, as this weight is increased ($W_T > 0.6$), the difference between the migration plans increases dramatically.

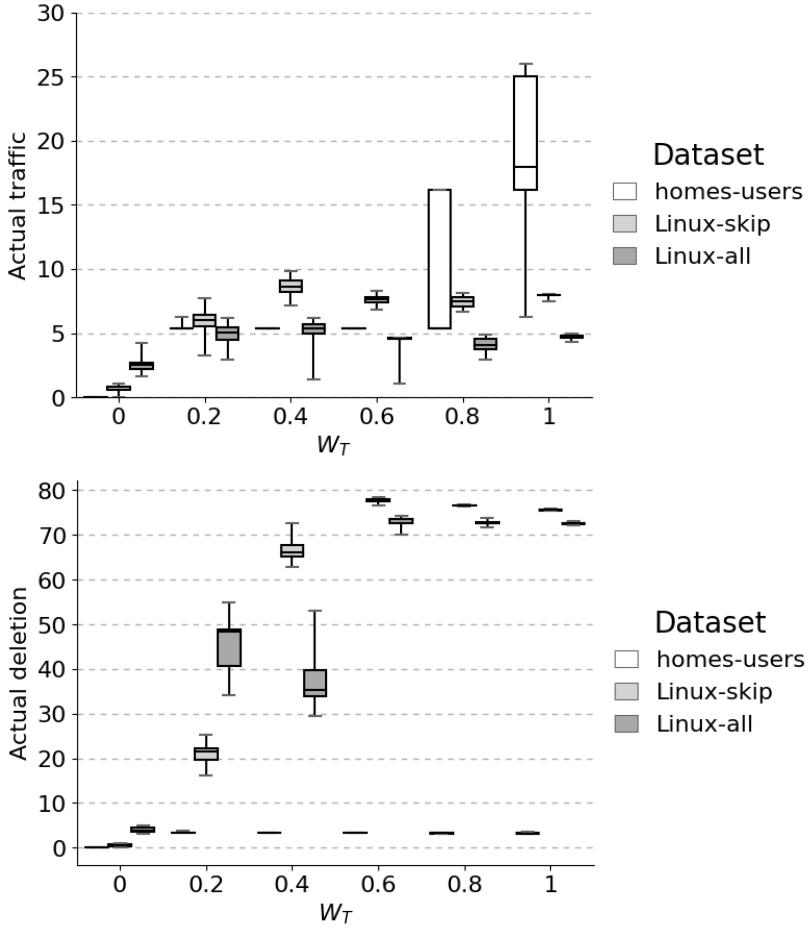


Figure 6.7: The distribution of migration traffic (top) and reduction in system size (bottom) in the set of plans returned by Cluster for different datasets with $k = 13$.

Effect of sampling degree. Figure 6.8 shows the deletion, load balancing, and runtime of all the algorithms on two samples of the Linux-skip system. The small and large samples were generated with sampling degrees of $k = 13$ and $k = 8$, respectively. The sample size affects each algorithm differently. Cluster returns similar results for both sample sizes because the differences in the accuracy of the sample are masked by its randomized process while greedy achieves a higher deletion on the larger sample (by up to 238%) and ILP suffers from the increase in the problem size. All the algorithms return a more balanced system for the larger sample ($k = 8$), because the load-balancing constraint is enforced on more blocks, and thus more accurately. At the same time, as we expected, their runtime was higher by several orders of magnitude, as the large sample included $2^5\times$ more blocks than the small one. As a result of those effects and in order to prevent those from happening, we chose to use randomization in our algorithm.

6.2.3 Sensitivity to problem parameters

Effect of load balancing and traffic constraints. Figure 6.9 shows the deletion, balance, and traffic consumption of all the algorithms on the UBC-500 system with

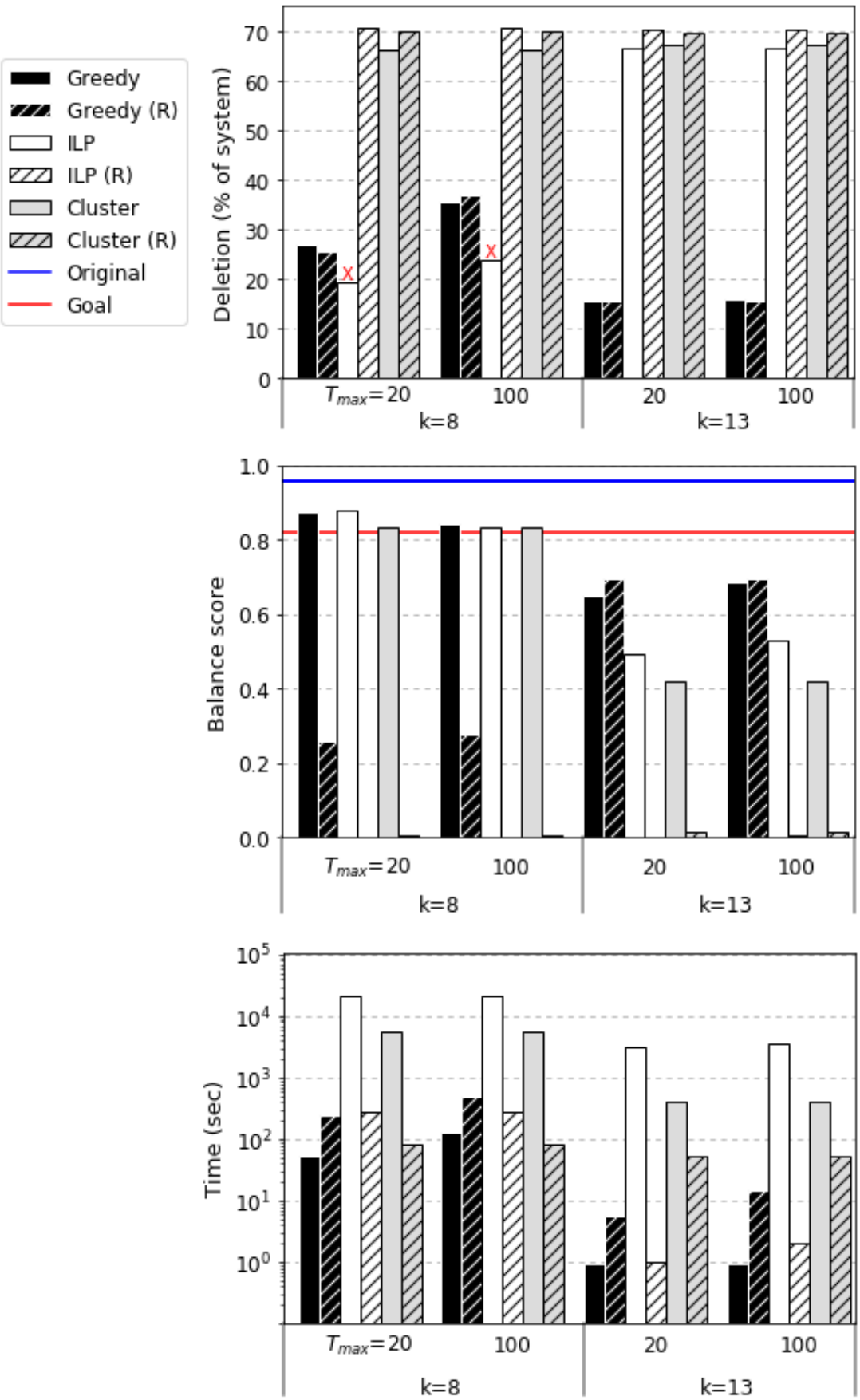


Figure 6.8: Linux-skip system with 5 volumes, $\mu = 2\%$, and two sampling degrees: $k = 8, 13$.

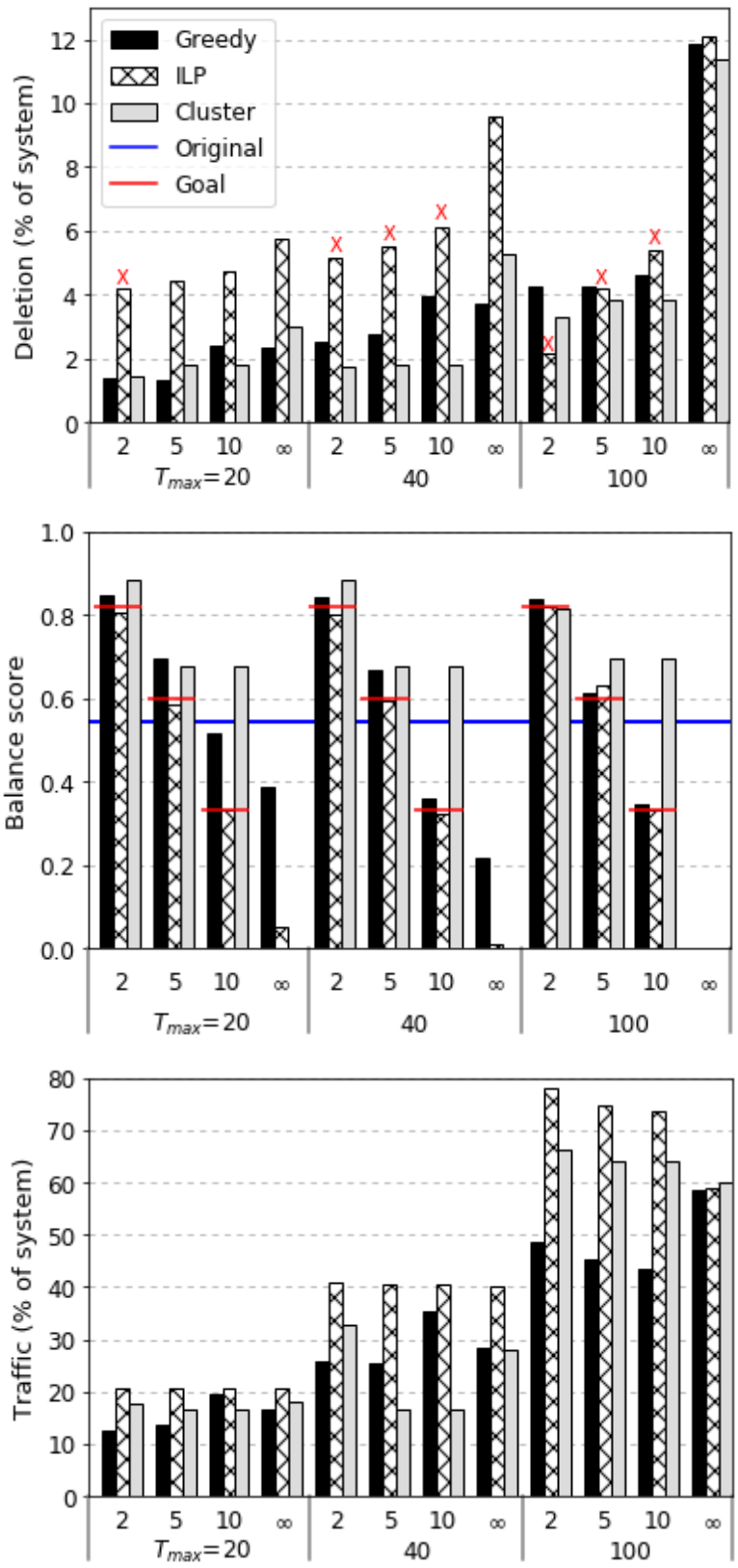


Figure 6.9: UBC-500 system with $k = 13$ and different load balancing margins.

different values of T_{max} and μ. The results on this system show the highest sensitivity to these constraints due to the relatively low similarity between its files. The deletion achieved by all the algorithms increases as T_{max} increases, and their traffic consumption increases accordingly. Removing the load-balancing constraint also allows for more deletion, as we observed in Figure 6.1. At the same time, the precise value of the load balancing margin, μ, has a much smaller effect on the achieved deletion, although in most cases, a lower margin does guarantee a more balanced system. As shown in Figure 6.10, increasing the margin usually increases the runtime of Greedy, as a result of more space-reduction iterations. The runtime of ILP and Cluster is not affected by the precise value of μ.

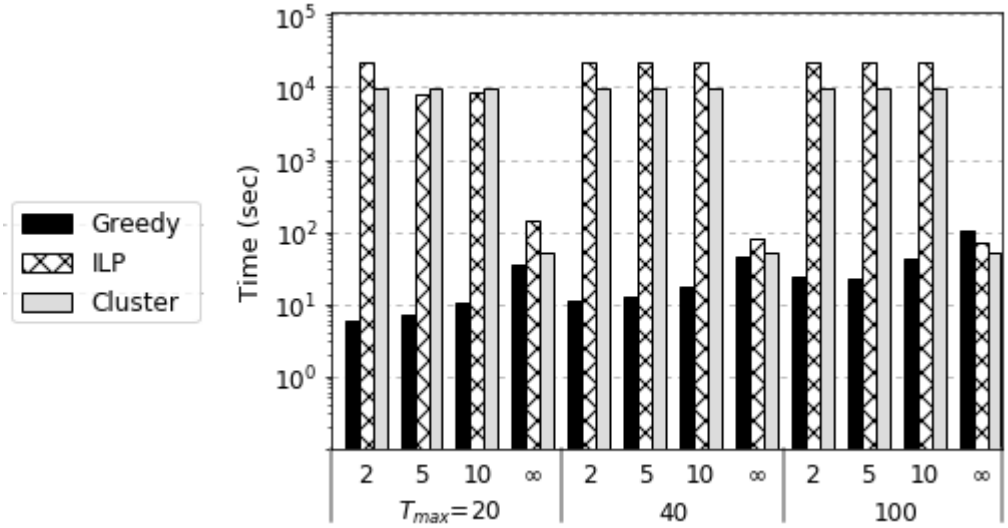


Figure 6.10: UBC-500 system with $k = 13$ and different load balancing margins - runtime

Effect of the number of volumes. Figure 6.11 shows the deletion and runtime of the algorithms on the Linux-skip system when the number of volumes is reduced ('4'), increased ('6'), or is larger overall ('10'). Due to the high similarity between the Linux versions, the same deletion is achieved when the number of volumes remains five, or when a volume is added or removed (the reduced performance of Cluster is an outlier for $\mu = 2\%$). When the initial number of volumes is 10, there are more duplicates in the system. This provides more opportunities for deletion, which is indeed higher.

The number of volumes affects the problem's complexity, affecting each algorithm differently. Greedy requires less time when a volume is added or removed (compared to a problem where the number of volumes remains the same) while the ILP problem complexity increases with every additional volume and thus its runtime increases until it reaches the timeout. The clustering process could, potentially, stop at an earlier stage when more clusters are needed. However, as the number of clusters increases the load balancing constraint is encountered at an earlier stage, causing the clustering to restart more often when the number of volumes is higher.

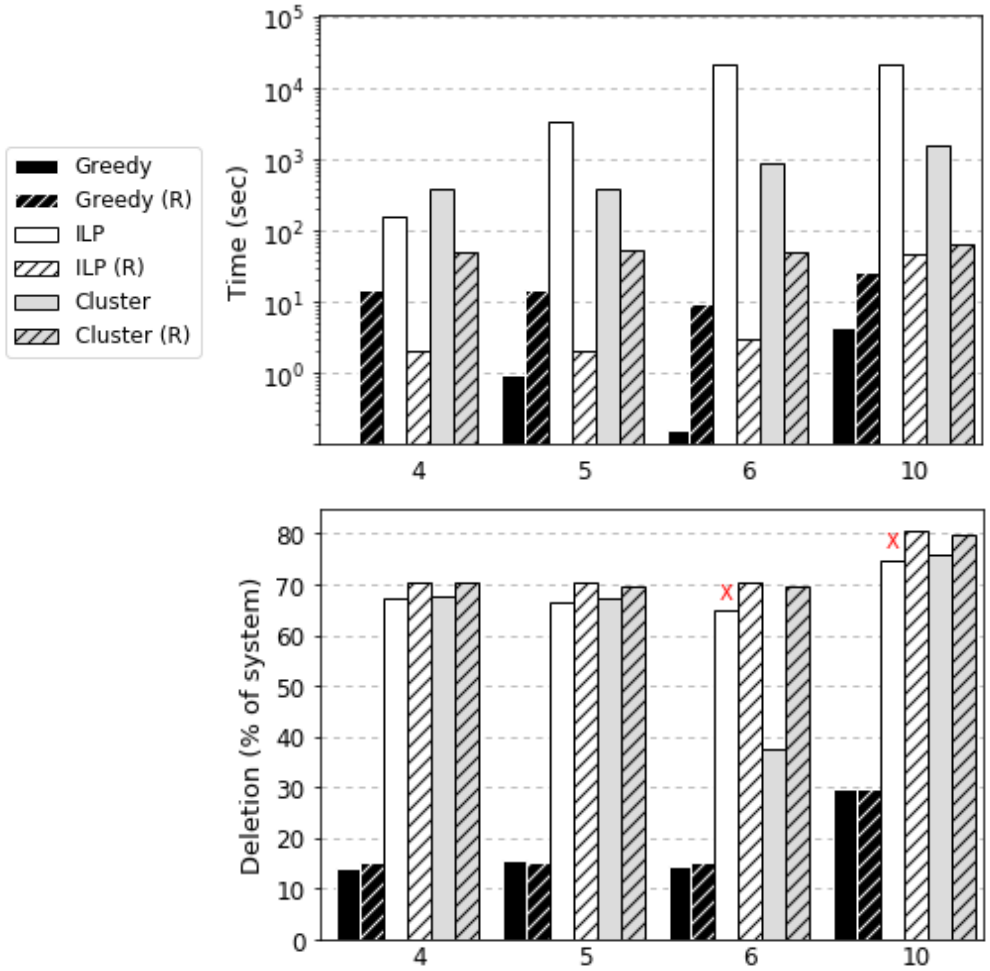


Figure 6.11: Linux-skip with different numbers of target volumes with $T_{max} = 100, k = 13, \mu = 2\%$.

Chapter 7

Conclusions

We formulated the general migration problem for storage systems with deduplication, and presented an algorithm for generating an efficient migration plan. Our evaluation showed that the greedy approach is the fastest but least effective, and that our clustering-based approach is comparable to the one based on ILP, despite ILP's premise of optimality. While the ILP-based approach guarantees a near-optimal solution (given sufficient runtime), clustering lends itself to a range of optimizations that enable it to produce such a solution faster.

Our approach can be applied to more specific cases of migration, presenting additional opportunities for further optimizations in the future. For example, thanks to its short runtime, we can use our algorithm to generate multiple plans with different traffic constraints. These plans are points on the Pareto frontier [ZKT08], i.e., they represent different tradeoffs between the conflicting objectives of maximizing deletion and minimizing traffic.

Bibliography

[AAA+10] Bhavish Aggarwal, Aditya Akella, Ashok Anand, Athula Balachandran, Pushkar Chitnis, Chitra Muthukrishnan, Ramachandran Ramjee, and George Varghese. EndRE: An end-system redundancy elimination service for enterprises. In *7th USENIX Conference on Networked Systems Design and Implementation (NSDI 10)*, 2010.

[ADK+18] Yamini Allu, Fred Douglis, Mahesh Kamat, Ramya Prabhakar, Philip Shilane, and Rahul Ugale. Can't we all get along? Redesigning protection storage for modern workloads. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, 2018.

[AHH+01] Eric Anderson, Joseph Hall, Jason D. Hartline, Michael Hobbs, Anna R. Karlin, Jared Saia, Ram Swaminathan, and John Wilkes. An experimental study of data migration algorithms. In *5th International Workshop on Algorithm Engineering (WAE 01)*, 2001.

[AHK+02] Eric Anderson, Michael Hobbs, Kimberly Keeton, Susan Spence, Mustafa Uysal, and Alistair Veitch. Hippodrome: Running circles around storage administration. In *1st USENIX Conference on File and Storage Technologies (FAST 02)*, 2002.

[BELL09] Deepavali Bhagwat, Kave Eshghi, Darrell D. E. Long, and Mark Lillibridge. Extreme binning: Scalable, parallel deduplication for chunk-based file backup. In *IEEE International Symposium on Modeling, Analysis Simulation of Computer and Telecommunication Systems (MASCOTS 09)*, 2009.

[BLC14] Bharath Balasubramanian, Tian Lan, and Mung Chiang. SAP: Similarity-aware partitioning for efficient cloud storage. In *IEEE Conference on Computer Communications (INFOCOM 14)*, 2014.

[CAVL09] Austin T. Clements, Irfan Ahmad, Murali Vilayannur, and Jinyuan Li. Decentralized deduplication in SAN cluster file systems. In *2009 Conference on USENIX Annual Technical Conference (USENIX 09)*, 2009.

[clu] Cluster analysis. `https://en.wikipedia.org/wiki/Cluster_analysis`. Accessed: 2020-10-24.

[CLZ11] Feng Chen, Tian Luo, and Xiaodong Zhang. CAFTL: A content-aware flash translation layer enhancing the lifespan of flash memory based solid state drives. In *9th USENIX Conference on File and Stroage Technologies (FAST 11)*, 2011.

[CPL] CPLEX Optimizer. https://www.ibm.com/analytics/cplex-optimizer. Accessed: 2018-10-24.

[DDL+11] Wei Dong, Fred Douglis, Kai Li, Hugo Patterson, Sazzala Reddy, and Philip Shilane. Tradeoffs in scalable data routing for deduplication clusters. In *9th USENIX Conference on File and Stroage Technologies (FAST 11)*, 2011.

[DDS+17] Fred Douglis, Abhinav Duggal, Philip Shilane, Tony Wong, Shiqin Yan, and Fabiano Botelho. The logic of physical garbage collection in deduplicating storage. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*, 2017.

[Ded22] DedupSearch: Two-Phase deduplication aware keyword search. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*, pages 233–246, Santa Clara, CA, February 2022. USENIX Association.

[DGH+09] Cezary Dubnicki, Leszek Gryz, Lukasz Heldt, Michal Kaczmarczyk, Wojciech Kilian, Przemyslaw Strzelczak, Jerzy Szczepkowski, Cristian Ungureanu, and Michal Welnicki. HYDRAstor: A scalable secondary storage. In *7th Conference on File and Storage Technologies (FAST 09)*, 2009.

[DJS+19] Abhinav Duggal, Fani Jenkins, Philip Shilane, Ramprasad Chinthekindi, Ritesh Shah, and Mahesh Kamat. Data Domain Cloud Tier: Backup here, backup there, deduplicated everywhere! In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, 2019.

[DSL10] Biplob Debnath, Sudipta Sengupta, and Jin Li. ChunkStash: Speeding up inline storage deduplication using flash memory. In *2010 USENIX Conference on USENIX Annual Technical Conference (USENIX ATC 10)*, 2010.

[FSL] Traces and snapshots public archive. http://tracer.filesystems.org/. Accessed: 2018-10-24.

[GE11] Fanglu Guo and Petros Efstathopoulos. Building a high-performance deduplication system. In *2011 USENIX Conference on USENIX Annual Technical Conference (USENIX ATC 11)*, 2011.

[Git] Source code of migration algorithms. `https://github.com/roei217/DedupMigration`. Accessed: 2022-02-22.

[GNU] GLPK (GNU Linear Programming Kit). https://www.gnu.org/software/glpk/. Accessed: 2018-10-24.

[GP13] Michael Greenacre and Raul Primicerio. *Hierarchical Cluster Analysis*. Fundación BBVA, Bilbao, 2013.

[Gur] The fastest mathematical programming solver. http://www.gurobi.com/. Accessed: 2018-10-24.

[GWM07] Ron Gabor, Shlomo Weiss, and Avi Mendelson. Fairness enforcement in switch on event multithreading. 4(3):15–es, September 2007.

[HHS+19] Danny Harnik, Moshik Hershcovitch, Yosef Shatsky, Amir Epstein, and Ronen Kat. Sketching volume capacities in deduplicated storage. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*, 2019.

[HKS16] Danny Harnik, Ety Khaitzin, and Dmitry Sotnikov. Estimating unseen deduplication-from theory to practice. In *14th Usenix Conference on File and Storage Technologies (FAST 16)*, 2016.

[HSX+12] Cheng Huang, Huseyin Simitci, Yikang Xu, Aaron Ogus, Brad Calder, Parikshit Gopalan, Jin Li, and Sergey Yekhanin. Erasure coding in Windows Azure Storage. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)*, 2012.

[KKD+22] Roei Kisous, Ariel Kolikant, Abhinav Duggal, Sarai Sheinvald, and Gala Yadgar. The *what*, the *from*, and the *to*: The migration games in deduplicated systems. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*, Santa Clara, CA, February 2022. USENIX Association.

[LAW02] Chenyang Lu, Guillermo A. Alvarez, and John Wilkes. Aqueduct: Online data migration with performance guarantees. In *1st USENIX Conference on File and Storage Technologies (FAST 02)*, 2002.

[LEB+09] Mark Lillibridge, Kave Eshghi, Deepavali Bhagwat, Vinay Deolalikar, Greg Trezise, and Peter Camble. Sparse indexing: Large scale, inline deduplication using sampling and locality. In *7th Conference on File and Storage Technologies (FAST 09)*, 2009.

[Lin] Linux Kernel Archives. `https://mirrors.edge.kernel.org/pub/linux/kernel/`.

[LLD+14] Xing Lin, Guanlin Lu, Fred Douglis, Philip Shilane, and Grant Wallace. Migratory compression: Coarse-grained data reordering to improve compressibility. In *12th USENIX Conference on File and Storage Technologies (FAST 14)*, 2014.

[lps] Introduction to lp_solve 5.5.2.5. http://lpsolve.sourceforge.net/5.5/. Accessed: 2018-10-24.

[LSD+14] Cheng Li, Philip Shilane, Fred Douglis, Hyong Shim, Stephen Smaldone, and Grant Wallace. Nitro: A capacity-optimized SSD cache for primary storage. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, 2014.

[Man94] Udi Manber. Finding similar files in a large file system. In *USENIX Winter 1994 Technical Conference (WTEC 94)*, 1994.

[MB11] Dutch T. Meyer and William J. Bolosky. A study of practical deduplication. In *9th USENIX Conference on File and Stroage Technologies (FAST 11)*, 2011.

[MCM01] Athicha Muthitacharoen, Benjie Chen, and David Mazières. A low-bandwidth network file system. In *18th ACM Symposium on Operating Systems Principles (SOSP 01)*, 2001.

[MHS18] Keiichi Matsuzawa, Mitsuo Hayasaka, and Takahiro Shinagawa. The quick migration of file servers. In *11th ACM International Systems and Storage Conference (SYSTOR 18)*, 2018.

[NEF+12] Edmund B. Nightingale, Jeremy Elson, Jinliang Fan, Owen Hofmann, Jon Howell, and Yutaka Suzue. Flat datacenter storage. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, 2012.

[NK13] P. C. Nagesh and Atish Kathpal. Rangoli: Space management in deduplication environments. In *6th International Systems and Storage Conference (SYSTOR 13)*, 2013.

[NSKY21] Aviv Nachman, Sarai Sheinvald, Ariel Kolikant, and Gala Yadgar. GoSeed: Optimal seeding plan for deduplicated storage. *ACM Trans. Storage*, 17(3), August 2021.

[RSG+13] K. V. Rashmi, Nihar B. Shah, Dikang Gu, Hairong Kuang, Dhruba Borthakur, and Kannan Ramchandran. A solution to the network challenges of data recovery in erasure-coded distributed storage systems: A study on the Facebook warehouse cluster. In *5th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 13)*, 2013.

[SBGV12] Kiran Srinivasan, Tim Bisson, Garth Goodson, and Kaladhar Voruganti. iDedup: Latency-aware, inline data deduplication for primary storage. In *10th USENIX Conference on File and Storage Technologies (FAST 12)*, 2012.

[SCJ16] Philip Shilane, Ravi Chitloor, and Uday Kiran Jonnala. 99 deduplication problems. In *8th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 16)*, 2016.

[SKM+16] Zhen Sun, Geoff Kuenning, Sonam Mandal, Philip Shilane, Vasily Tarasov, Nong Xiao, and Erez Zadok. A long-term user-centric analysis of deduplication patterns. In *32nd Symposium on Mass Storage Systems and Technologies (MSST 16)*, 2016.

[SNI] SNIA IOTTA Repository. http://iotta.snia.org/tracetypes/6. Accessed: 2018-10-24.

[SYM] SYMPHONY development home page. https://projects.coin-or.org/SYMPHONY. Accessed: 2018-10-24.

[TAB11] Nguyen Tran, Marcos K. Aguilera, and Mahesh Balakrishnan. Online migration for geo-distributed storage systems. In *2011 USENIX Conference on USENIX Annual Technical Conference (USENIX ATC 11)*, 2011.

[TMB+12] Vasily Tarasov, Amar Mudrankit, Will Buik, Philip Shilane, Geoff Kuenning, and Erez Zadok. Generating realistic datasets for deduplication analysis. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)*, 2012.

[WBMM06] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, and Carlos Maltzahn. CRUSH: Controlled, scalable, decentralized placement of replicated data. In *ACM/IEEE Conference on Supercomputing (SC 06)*, 2006.

[XJF+14] Wen Xia, Hong Jiang, Dan Feng, Lei Tian, Min Fu, and Yukun Zhou. Ddelta: A deduplication-inspired fast delta compression approach. *Performance Evaluation*, 79:258 – 272, 2014. Special Issue: Performance 2014.

[XZJ+16] Wen Xia, Yukun Zhou, Hong Jiang, Dan Feng, Yu Hua, Yuchong Hu, Qing Liu, and Yucheng Zhang. FastCDC: A fast and efficient content-defined chunking approach for data deduplication. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, 2016.

[YJTL16] Zhichao Yan, Hong Jiang, Yujuan Tan, and Hao Luo. Deduplicating compressed contents in cloud storage environment. In *8th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 16)*, 2016.

[ZKT08] Eckart Zitzler, Joshua Knowles, and Lothar Thiele. *Quality Assessment of Pareto Set Approximations*, pages 373–404. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[ZLP08] Benjamin Zhu, Kai Li, and Hugo Patterson. Avoiding the disk bottleneck in the Data Domain deduplication file system. In *6th USENIX Conference on File and Storage Technologies (FAST 08)*, 2008.

אנו מציעים דרך לבניית תוכנית מיגרציה יעילה בעזרת אלגוריתם קיבוץ היררכי (hierarchical clustering). קיבוץ מתייחס לשיוך אובייקטים לקבוצות על פי הדמיון ביניהם. בפרט, קיבוץ היררכי, מתייחס למרחק בין האובייקטים. בהתחלה, כל אובייקט ממופה לקבוצה בפני עצמו והתהליך האיטרטיבי מאחד בכל שלב את שתי הקבוצות עם המרחק המינימלי ביניהן. אנו מעוניינים בקיבוץ קבצים דומים כדי להפחית את כמות המידע הפיזי במערכת תוך מתן חשיבות לניצול נמוך של תעבורת הרשת ואיזון המערכת. כשתהליך הקיבוץ מסתיים, אנו מחשבים את כמות המידע הפיזי שניתן למחוק, תעבורת הרשת שנוצלה ואיזון המערכת שהתקבל כדי לקבוע האם מדובר בתוכנית מיגרציה טובה או לא.

לצורך שערוך של ביצועי האלגוריתם יצרנו שישה מקבצי גיבויים המייצגים מערכות אמתיות: שני מקבצי גיבויים של מערכות לינוקס (Linux), מקבץ גיבויים של מערכות מייקרוסופט (Microsoft) ושני מקבצי גיבויים של סטודנטים במעבדת FSL באוניברסיטת סטוני ברוק (Homes). ביצענו מספר ניסויים שכללו את כל מקבצי הגיבויים עם פרמטרים שונים לבעיה על מנת לנתח את ביצועי האלגוריתם.

התוצאות מראות שבשיטה זו יש מספר שקלולי תמורות בין זמן החישוב ויעילות תוכנית המיגרציה אל מול אלגוריתמים אחרים כמו האלגוריתם החמדן (Greedy) ואלגוריתם תכנון לינארי (ILP). האלגוריתם שלנו משיג תוצאות זהות ולעתים אף טובות מהאלגוריתם לתכנון לינארי, שהוא באופן תיאורטי, אופטימלי, אך בפחות זמן.

תקציר

דדופליקציה (deduplication) היא שיטה מובילה להפחתת גודל האחסון כאשר קיימת כפילות מידע. שיטה זו יכולה לפעול על בלוקים, קבצים, קונטיינרים וכו'. במקום לשמור מספר עותקים של אותו מידע, שומרים קישור בין המידע הלוגי לעותק פיזי בודד ובכך נמנע שכפול מידע מיותר. עקב זאת, מידע משותף בין אובייקטים כמו קבצים או תיקיות שלמות וגורם לתקורה באיסוף האשפה, שליפת מידע מדויק על המערכת ואתגרים במיגרציה (הגירה).

מחקרים אקדמיים קודמים ומערכות מסחריות כבר בחנו לעומק ושיפרו את הסיבוכיות של קריאה, כתיבה ומחיקה של מידע במערכות עם דדופליקציה, אך עדיין ישנם כמה היבטי ניהול של מערכות גדולות עם דדופליקציה שטרם נבחנו כמו שמירה במטמון, תכנון תוכן המערכת ועלויות נוספות.

בעבודה זו, אנו מתייחסים לבעיית המיגרציה הכללית שבה קבצים ממופים בין שרתים שונים עקב הרחבת המערכת או תחזוקה. השאלה אלו קבצים ובלוקים להעביר, ולאן, נחקרה לא מעט במערכות ללא דדופליקציה אך רק מקרים פשוטים נבדקו בהקשר של מערכות עם דדופליקציה.

עקב המידע המשותף בין הקבצים, בחירת קבצים ובלוקים אלו צריכה להתייחס לכמה היבטים: כאשר קובץ מועבר, חלק מהבלוקים שלו יכולים להימחק משרת המקור, בעוד בלוקים שמשותפים לקובץ זה אך גם לקבצים אחרים צריכים להיות משוכפלים על שרתי המקור והיעד. באופן דומה, חלק מהבלוקים של אותו הקובץ יכולים להיות מועברים לשרת היעד, בעוד חלק אחר יכול להיות כבר קיים בשרת זה ולכן אין צורך בהעברתם.

כחלק מתוכנית המיגרציה הכללית, מטרתנו היא להקטין ככל הניתן את גודל המערכת תוך איזון כמות המידע הפיזי בין השרתים השונים במערכת והפחתת כמות תעבורת הרשת העוברת כחלק מן תהליך המיגרציה.

מספר מחקרים תיארו בעבר מקרים ספציפיים של מיגרציה. באחד מהם נדרש למזער את גודל המערכת הכולל, בעוד באחר נדרש למחוק קבצים על מנת להקטין את גודל המערכת הכללי. במחקרים מאוחרים יותר שבוצעו נעשה שימוש בתכנון לינארי (ILP) על מנת לפתור את בעיית הזריעה (seeding) בה קבצים מועברים משרת אחד לשרת ריק אחר ולאחר מכן כדי לפתור את בעיית המיגרציה הכללית שעבורה גם הוצע אלגוריתם חמדן המוצא תוכנית מיגרציה כללית גם כן.

המחקר בוצע בהנחייתה של דוקטור גלה ידגר, בפקולטה למדעי המחשב על שם הנרי ומרלין טאוב.

חלק מן התוצאות בחיבור זה פורסמו כמאמרים מאת המחבר ושותפיו למחקר בכנסים ובכתבי-עת במהלך תקופת המחקר של המחבר, אשר גרסאותיהם העדכניות ביותר הינן:

Roei Kisous, Ariel Kolikant, Abhinav Duggal, Sarai Sheinvald, and Gala Yadgar. The *what*, the *from*, and the *to*: The migration games in deduplicated systems. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*, 2022.

Roei Kisous, Ariel Kolikant, Abhinav Duggal, Sarai Sheinvald, and Gala Yadgar. The *what*, the *from*, and the *to*: The migration games in deduplicated systems. Invited and submitted to the Special Section on FAST22 in the Transactions on Storage.

תודות

ברצוני להודות לד"ר גלה ידגר על השקעת הזמן והמאמץ שעלו על הציפיות שלי. הרעיונות שתרמת, תהליך סיעור המוחות שנעשה בו שימוש, ההדרכה המצוינת שנתת, ההשראה שנתת והתהליך הנעים שהובלת עשו שינוי משמעותי בתוצאה.

אני גם רוצה להודות לאבינב דוגל על העצות והרעיונות המצוינים שלו, כמו גם על מתן נקודת מבט מעניינת מהתעשייה.

על תמיכתם ואהבתם ללא תנאי, אני מודה להוריי, מירב ויוסי במיוחד. לבסוף, אני רוצה להודות לכל החברים שלי על התמיכה והחברות, שהפכו את החוויה להרבה יותר עשירה.

אני מודה לטכניון על התמיכה הכספית הנדיבה בהשתלמותי.

המחקר נתמך ע"י הקרן למדע של ישראל (מענק מספר 807/20).

הגירת נתונים מבוססת קיבוץ במערכות אחסון עם דדופליקציה

חיבור על מחקר

לשם מילוי חלקי של הדרישות לקבלת התואר
מגיסטר למדעים במדעי המחשב

רואי קיסוס

הוגש לסנט הטכניון — מכון טכנולוגי לישראל
תמוז התשפ"ב חיפה יולי 2022

הגירת נתונים מבוססת קיבוץ במערכות אחסון עם דדופליקציה

רואי קיסוס