The *what*, The *from*, and The *to*: The Migration Games in Deduplicated Systems

Roei Kisous and Ariel Kolikant, *Technion - Israel Institute of Technology;*
Abhinav Duggal, *DELL EMC;* Sarai Sheinvald, *ORT Braude College of Engineering;*
Gala Yadgar, *Technion - Israel Institute of Technology*

https://www.usenix.org/conference/fast22/presentation/kisous

This paper is included in the Proceedings of the
20th USENIX Conference on File and Storage Technologies.
February 22–24, 2022 • Santa Clara, CA, USA
978-1-939133-26-7

The *what*, The *from*, and The *to*: The Migration Games in Deduplicated Systems

Roei Kisous and Ariel Kolikant
Computer Science Department, Technion

Abhinav Duggal
DELL EMC

Sarai Sheinvald
ORT Braude College of Engineering

Gala Yadgar
Computer Science Department, Technion

Abstract

Deduplication reduces the size of the data stored in large-scale storage systems by replacing duplicate data blocks with references to their unique copies. This creates dependencies between files that contain similar content, and complicates the management of data in the system. In this paper, we address the problem of data migration, where files are remapped between different volumes as a result of system expansion or maintenance. The challenge of determining which files and blocks to migrate has been studied extensively for systems without deduplication. In the context of deduplicated storage, however, only simplified migration scenarios were considered.

In this paper, we formulate the general migration problem for deduplicated systems as an optimization problem whose objective is to minimize the system's size while ensuring that the storage load is evenly distributed between the system's volumes, and that the network traffic required for the migration does not exceed its allocation.

We then present three algorithms for generating effective migration plans, each based on a different approach and representing a different tradeoff between computation time and migration efficiency. Our *greedy algorithm* provides modest space savings, but is appealing thanks to its exceptionally short runtime. Its results can be improved by using larger system representations. Our *theoretically optimal algorithm* formulates the migration problem as an ILP (integer linear programming) instance. Its migration plans consistently result in smaller and more balanced systems than those of the greedy approach, although its runtime is long and, as a result, the theoretical optimum is not always found. Our *clustering algorithm* enjoys the best of both worlds: its migration plans are comparable to those generated by the ILP-based algorithm, but its runtime is shorter, sometimes by an order of magnitude. It can be further accelerated at a modest cost in the quality of its results.

1 Introduction

Many large-scale storage systems employ data deduplication to reduce the size of the data that they store. The deduplication process identifies duplicate data blocks in different files and replaces them with pointers to a unique copy of the block stored in the system. This reduction in the system's size comes at the cost of increased system complexity. While the complexity of reading, writing, and deleting data in deduplicated storage systems has been addressed by many academic studies and commercial systems, the high-level management aspects of large-scale systems, such as capacity planning, caching, and quality and cost of service, still need to be adapted to deduplicated storage [44].

This paper focuses on the aspect of *data migration*, where files are remapped between separate deduplication domains, or *volumes*. A volume may represent a single server within a large-scale system, or an independent set of servers dedicated to a customer or dataset. Files might be remapped as a result of volumes reaching their capacity limitation or of other bottlenecks forming in the system. Deduplication introduces new considerations when choosing which files to migrate, due to the data dependencies between files: when a file is migrated, some of its blocks may be deleted from its original volume, while others might still belong to files that remain on that volume. Similarly, some blocks need to be transferred to the target volume, while others may already be stored there. An efficient migration plan must optimize several, possibly conflicting objectives: the physical size of the stored data after migration, the load balancing between the system's volumes, i.e., the physical size of the data stored on each volume, and the network bandwidth generated by the migration itself.

Several recent studies address specific (simplified) cases of data migration in deduplicated systems. Harnik et al. [28] address capacity estimation and propose a greedy algorithm for reducing the system's size. Rangoli [41] is a greedy algorithm for *space reclamation*, where a set of files is deleted to reclaim some of the system's capacity. GoSeed [40] is an ILP (integer linear programming)-based algorithm for the *seeding* problem, in which files are remapped into an initially empty target volume. While even the seeding problem is shown to be NP-hard [40], none of these studies address the conflicting objectives involved in the full data migration problem. Namely, the tradeoff between minimizing the system size, minimizing the network traffic consumed during migration, and maximizing the load balance between the volumes in the system.

In this paper, we address, for the first time, the general case of data migration. We begin by formulating the data migration problem in its most general form, as an optimization problem whose main goal is to minimize the overall size of the system. We add the traffic and load balancing considerations as constraints on the migration plan. The degree in which these constraints are enforced directly affects the solution space, allowing the system administrator to prioritize different costs. Thus, the problem of data migration in deduplication systems maps to finding what to migrate, where to migrate from, and where to migrate to within the traffic and load balancing constraints specified by the administrator.

We then introduce three novel algorithms for generating an efficient migration plan. The first is a greedy algorithm that is inspired by the greedy iterative process in [28]. Our extended algorithm distributes the data evenly between volumes while ensuring that the migration traffic does not exceed the maximum allocation. By breaking this process into several phases, we ensure that the allocated traffic is used for both load balancing and capacity reduction, balancing between the two (possibly conflicting) goals.

Our second algorithm is inspired by the ILP-based approach of GoSeed. GoSeed solves the seeding problem, whose single natural minimization objective is the system size. In contrast, our new algorithm addresses the inherently competing objectives (size, balance, traffic) of general migration. We reformulate the ILP problem with variables and constraints that express the traffic used during migration and the choice of volumes from which to remap files or to remap files onto. Our formulation for the general migration problem is naturally much more complex than the one required for seeding. Nevertheless, we successfully applied it to data migration in systems with hundreds of millions of blocks.

Our third algorithm is based on hierarchical clustering, which, to the best of our knowledge, has not been applied to data deduplication before. We group similar files into clusters, where the target number of clusters is the number of volumes in the system. We incorporate the physical location of the files into the clustering process, such that the similarity between files expresses the blocks that they share as well as their initial locations. Clusters are assigned to volumes according to the blocks already stored on them, and the migration plan remaps each file to the volume assigned to its cluster.

We implemented our three algorithms and evaluated them on six system snapshots created from three public datasets [6, 10, 38]. Our results demonstrate that all algorithms can successfully reduce the system's size while complying with the traffic and load balancing constraints. Each algorithm has different advantages: the greedy algorithm produces a migration plan in the shortest runtime (often several seconds), although its reduction in system size is typically lower than that of the other algorithms. The ILP-based approach can efficiently utilize the allowed traffic consumption, and improve as the load balancing constraints are relaxed. However, its execution must be timed out on the large problem instances, which often prevents it from yielding an optimal migration plan. The clustering algorithm empirically achieves comparable results to those of the ILP-based approach, and sometimes even exceeds them. It does so in much shorter runtimes.

We summarize our main contributions as follows. We formulate the general migration with deduplication as an optimization problem (§ 3), and design and implement three algorithms for generating general migration plans: the greedy (§ 4) and ILP-based (§ 5) approaches are inspired by previous studies, while the clustering-based (§ 6) approach is entirely novel. We methodologically compare our algorithms to analyze the advantages and limitations of each approach (§ 7).

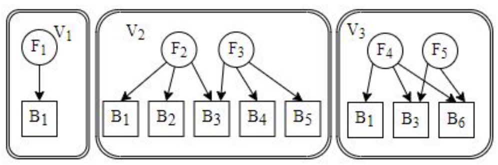
2 Background and related work

Data deduplication. In a nutshell, the deduplication process splits incoming data into fixed or variable-sized chunks, which we refer to as *blocks*. The content of each block is hashed to create a *fingerprint*, which is used to identify duplicate blocks and to retrieve their unique copy from storage. Several aspects of this process must be optimized so as not to interfere with storage system performance. These include chunking and fingerprinting [11,36,39,50,51], indexing and lookups [12,45,54], efficient storage of blocks [17, 19, 31, 33, 34, 45, 52], and fast file reconstruction [24, 30, 32, 53]. Although the first commercial systems used deduplication for backup and archival data, deduplication is now commonly used in high-end primary storage.

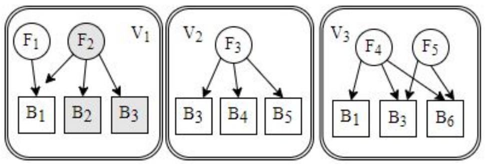
Data migration in distributed deduplication systems. Numerous distributed deduplication designs were introduced in commercial and academic studies [18, 22, 27]. We focus on designs that employ a separate fingerprint index in each physical server [15, 16, 20, 21, 28]. This design choice maintains a small index size and a low lookup cost, facilitates garbage collection at the server level, and simplifies the client-side logic. In this design, each server (*volume*) is a separate *deduplication domain*, i.e., duplicate blocks are identified only within the same volume. Recipes of files mapped to a specific volume thus point to blocks that are physically stored in that volume.

Deduplicated systems are different from traditional distributed systems in that striping files across volumes might reduce deduplication, even if it is done using a content-based chunking algorithm. Splitting files across a cluster also complicates garbage collection. Moreover, many storage systems (e.g., in DataDomain [23] and IBM [28]) are organized as a collection of independent clusters or "islands" of storage in the data center or across data centers. Deduplication is performed within each independent subsystem, but files might be migrated between the different appliances or clusters as a means to re-balance the entire system's utilization.

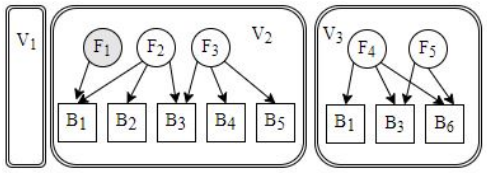
For example, if a subsystem becomes full while another subsystem has available capacity, migration is quicker and cheaper than adding capacity to the full subsystem. Existing mechanisms migrate files efficiently by transferring only the files' metadata and the chunks that are not already present in the target subsystem [23]. Monthly migration aligns with average



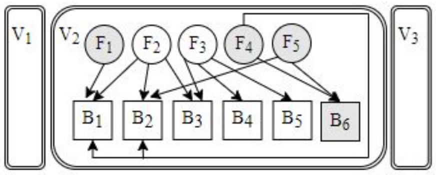
(a) Initial system: balance = 1/5



(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 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.

retention period which is seen for typical backup customers.

The coupling of the logical file's location and the physical location of its blocks implies that when a file is remapped from its volume, we must ensure that all its blocks are stored in the new volume. At the same time, the file's blocks cannot necessarily be removed from its original volume, because they might also belong to other files. For example, consider the initial system depicted in Figure 1(a), and assume we remap file F_2 from volume V_2 to volume V_1, resulting in the alternative system in Figure 1(b). Block B_1 is deleted from V_2 because it is already stored in V_1. Block B_2 is deleted from V_2, but must be copied to V_1, because it wasn't there in the initial system. Block B_3 must also be copied to V_1, but is not deleted from V_2 because it also belongs to F_3. The total sizes of the initial system and of this alternative are the same: nine blocks.

Existing approaches. Harnik et al. [28] presented a greedy iterative algorithm for reducing the total capacity of data in a system with multiple volumes. In each iteration, one file is remapped to a new volume, and the process continues until the total capacity is reduced by a predetermined deletion goal.

GoSeed [40] addresses a simplified case of data migration called *seeding*, where the initial system consists of many files mapped to a single volume. The migration goal is to delete a portion of this volume's blocks by remapping files to an empty target volume [23]. GoSeed formulates the seeding problem as an ILP (integer linear programming) instance whose solution determines which files are remapped, which blocks are *moved* from the source volume to the target, and which are *replicated* to create copies on both volumes. This approach is made possible by the existence of open-source [4, 5, 9] and commercial [2, 3] ILP-solvers—heuristic-based software tools for solving this NP-hard problem efficiently. GoSeed is applied to instances with millions of blocks with several acceleration heuristics, some of which we adapt to the generalized problem.

Rangoli [41] is a greedy algorithm for *space reclamation*—another specific case of data migration where a set of files is chosen for deletion in order to delete a portion of the system's physical size. Unlike the greedy and ILP-based approaches that inspire our own algorithms, the problem solved by Rangoli is too simplified for it to be extended for general migration. Shilane et al. [44] discuss additional data migration scenarios and their resulting complexities in deduplicated systems.

3 Motivation and problem statement

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 [29, 43]. 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 [13, 14, 23, 35, 37, 48]. 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. [28] did not address this aspect at all. In the seeding problem addressed by GoSeed [40], 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 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 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.

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 [14, 42, 49]. 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 [16, 20]. 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. All our approaches can be extended to address it explicitly.

In this work, we capture the load balancing in the system with the *balance* metric, which is similar to a commonly used *fairness* metric [25]—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 1(a) is $|V_1|/|V_2| = 1/5$. Alternative 1 (Figure 1(b)) is perfectly balanced, with $balance = 1$, while Alternative 2 (Figure 1(c)) has the worst balance: $|V_1|/|V_2| = 0$.

Problem statement. There are various approaches for handling conflicting objectives in complex optimization systems. These include searching for the Pareto frontier [55], 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 [40], 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 $balance \leq [size(S')/|V| \times (1-\mu)] / [size(S')/|V| \times (1+\mu)]$.

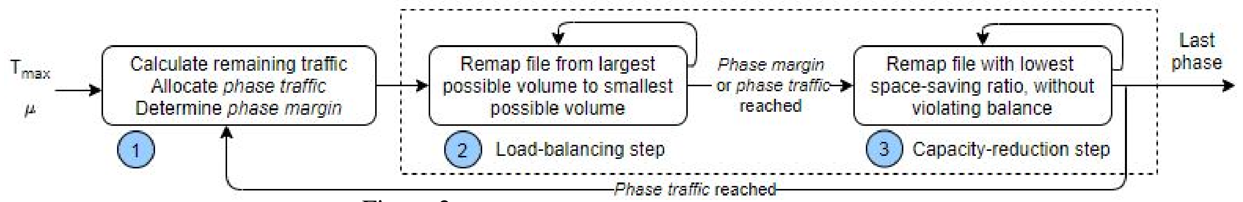
For example, for the initial system in Figure 1(a), Alternative 1 (Figure 1(b)) is the only migration plan that satisfies the load balancing constraint (for any μ). For T_{max} lower than $2/9$, no migration is feasible. On the other hand, if we remove the load balancing constraint, the optimal migration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

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.

4 Greedy

The basic greedy algorithm by Harnik et al. [28] iterates over all the files in each volume, and calculates the *space-saving ratio* from remapping a single file to each of the other volumes: the ratio between the total size of the blocks that would be replicated and the blocks that would be deleted from the file's original volume. In each iteration, the file with the lowest ratio is remapped. For example, if this basic greedy algorithm was applied to the initial system in Figure 1(a), it would first remap file F_1 to volume V_2, with a space-saving ratio of 0, resulting in Alternative 2 (Figure 1(c)). The process halts when the total capacity is reduced by a predetermined deletion goal. This algorithm is not directly applicable to the general migration problem because it does not consider traffic and load balancing.

Addressing the traffic constraint is relatively straightforward. In our extended greedy algorithm we make it the halting condition: the iterations stop when there is no file that can be



T_max
μ
Calculate remaining traffic Allocate phase traffic Determine phase margin
1
Remap file from largest possible volume to smallest possible volume
2 Load-balancing step
Phase margin or phase traffic reached
Remap file with lowest space-saving ratio, without violating balance
3 Capacity-reduction step
Last phase
Phase traffic reached

Figure 2: Overview of our extended greedy algorithm.

remapped without exceeding the maximum allocated traffic. A small challenge is that a file might be remapped in several iterations of the algorithm, while, in the resulting migration plan, it will only be remapped from its original volume to its final destination. As a result, the sum of traffic of all the individual iterations can be (and is, in practice) higher than the traffic required when executing migration plan. This will not violate the traffic constraint, but will cause the algorithm to halt before taking advantage of the maximum allowed traffic. Thus, we heuristically allow the algorithm to use 20% more traffic than the original traffic constraint, to prevent it from halting prematurely. The required traffic for the resulting migration plan is calculated before its execution. Thus, if it violates the original traffic constraint, a new plan can be generated by the algorithm without this heuristic. We include this simple extension, without a load-balancing constraint, in our evaluation.

Complying with the load-balancing constraint is more challenging. For example, if the basic greedy algorithm reached Alternative 2 (Figure 1(c)), it could no longer remap any single file to volume V_1 without increasing the system's capacity, and thus the system will remain unbalanced with at least one empty volume. A naive extension to this algorithm could enforce the load-balancing constraint by preventing files from being remapped if this increases the system's imbalance. However, such a strict requirement might preclude too many opportunities for optimization. For example, for the initial system in Figure 1(a), it would only allow to remap file F_2 to volume V_1, resulting in Alternative 1 (Figure 1(b)). The system would be perfectly balanced, but the basic algorithm would then terminate without reducing its size at all.

We address this challenge with two main techniques. The first is defining two iteration types: one whose goal is to balance the system's load, and another whose goal is to reduce its size. We perform these iterations interchangeably, to avoid the entire allocated traffic from being spent on only one goal. The second technique is to relax the load-balancing margin for the early iterations and continuously tighten it until the end of the execution. The idea is to let the early iterations remap files more freely, and to ensure that the iterations at the end of the algorithm result in a balanced system.

Figure 2 illustrates the process of our extended greedy algorithm. We divide the algorithm's process into phases. ① Each phase is allocated an even portion of the traffic allocated for migration, and is limited by a local load-balancing constraint. Each phase is composed of two steps. ② The *load-balancing step* iteratively remaps files from large volumes to small ones, until the volume sizes are within the margin defined for this phase, or its traffic is exhausted. ③ The *capacity-reduction step* uses the remaining traffic to reduce the system's size by remapping files between volumes, ensuring that volume sizes remain within the margin.

Each phase is limited by **local traffic and load-balancing constraints**, calculated at the beginning of the phase. The *phase traffic* determines the maximum traffic that can be used in each phase, and is roughly even for all the phases. The local *phase margin* determines the minimum and maximum allowed volume sizes in each phase. It is larger than the global margin, μ, in the first phase, and gradually decreases before each phase, until reaching μ in the last phase. By default, our greedy algorithm consists of $p = 5$ phases. The phase traffic for phase i, $0 \le i < p$, is $1/(p-i)$ of the unused traffic, and the phase margin for the first phase is $\mu \times 1.5$.

The load balancing step is the first step in each phase. In each of its iterations, the volumes are sorted according to their sizes, and we attempt to remap files from the largest volumes to the small ones. A file can be remapped only if some blocks will be deleted from its source volume as a result. Namely, we look for a file to remap between a $\langle source, target\rangle$ pair of volumes, where *source* is the largest volume and the *target* is the smallest volume for which such a file exists. In each iteration, the amount of traffic required to remap the chosen file is calculated, and the iterations halt when the maximum allowed traffic or allowed volume sizes are reached.

The capacity-reduction step uses the remaining traffic allocation of the phase. It is similar to the original greedy algorithm, but it ensures that each file remap does not cause the volumes to become unbalanced. In other words, we can remap a file only if this does not cause its source volume to become too small, or its target volume to become too large. Note that the amount of traffic that remains for the capacity-reduction step depends on the degree of imbalance in the initial system. In the most extreme case of a highly unbalanced system, it is possible for the load balancing step to consume all the traffic allocated for the phase. In this case, the capacity-reduction step halts in the first iteration. For cases other than this extreme, a higher number of phases can divert more traffic for capacity-reduction, at the cost of longer computation time due to the increased number of iterations.

5 ILP

Our ILP-based approach is inspired by GoSeed [40], designed for the seeding problem, where files can only be remapped from the source volume to the empty target volume. GoSeed thus defined three types of variables whose assignment specified (1) whether a file is *remapped*, (2) whether a block is *replicated* on both volumes, and (3) whether a block is deleted from the source and *moved* to the target. These limited options resulted in a fairly simple set of constraints, which cannot be directly applied to the general migration problem. The major difference is that the decision of whether or not we can delete a block from its source volume depends not only on the files initially mapped to this volume, but also on the files that will be remapped to it as a result of the migration. Thus, in our ILP-based approach, every block transfer is modeled as creating a copy of this block, and a separate decision is made whether or not to delete the block from its source volume.

The problem's constraints are defined over the set of volumes, files, and blocks from the problem statement in Section 2, the maximum traffic T_{max}, and the load-balancing margin μ. We define the target size of each volume v as w_v, given as percentage of the system size after migration. By default, $w_v = 1/|V|$. For each pair of volumes, v, u, we define their *intersection* as the set of blocks that are stored on both volumes: $Intersect_{vu} = \{b | b \in u \wedge b \in v\}$. The intersections are calculated before the constraints are assigned, and are used in the formulation below for better readability.

The constraints are expressed in terms of three types of variables that denote the actions performed in the migration: x_{fst} denotes whether file f is *remapped* from its source volume s to another (target) volume t. c_{bst} denotes whether block b is *copied* from its source volume s to another (target) volume t. Finally, d_{bv} denotes whether block b is *deleted* from volume v. The solution to the ILP instance is an assignment of 0 or 1 to these variables. The resulting migration plan remaps the set of files for which $x_{fst} = 1$ (for some volume t), transfers the blocks for which $c_{bst} = 1$ to their target volume, and deletes the blocks for which $d_{bv} = 1$ from their respective volumes.

Constraints and objective. The ILP formulation for migration with load balancing consists of 13 constraint types.

1. All Variables are Boolean.
2. A file can be remapped to at most one volume.
3. A block can only be deleted or copied from a volume it was originally stored in.
4. A block can be deleted from a volume only if all the files containing it are remapped to other volumes.
5. A block can be deleted from a volume only if no file containing it is remapped to this volume.
6. View all the blocks in the volume intersections as having been copied.
7. When a file is remapped, all its blocks are either copied to the target volume, or are initially there (as part of the intersection).
8. A block can be copied to a target volume only from one source volume.and volume t, $\Sigma_{s \text{ such that } b \notin Intersect_{st}} c_{bst} \leq 1$.
9. A block must be deleted if there are no files containing it on the volume.
10. *A* block cannot be copied to a target volume if no file will contain it there.
11. A file cannot be migrated to its initial volume.
12. *Traffic constraint*: the size of all the copied blocks is not larger than the maximum allowed traffic.
13. *Load balancing constraint*: for each volume v, $(w_v - \mu) \times Size(S') \leq Size(v') \leq (w_v + \mu) \times Size(S')$, where $Size(v')$ is the volume size after migration, i.e., the sum of its non-deleted blocks and blocks copied to i.

► *Objective*: maximize the sum of sizes of all blocks that are deleted minus all blocks that are copied. This is equivalent to minimizing the overall system size.

Constraints 12 and 13 formulate the traffic and load-balancing goals, and constraints 8, 9, and 10 ensure that the solver does not create redundant copies of blocks to artificially comply with the load balancing constraint. This is similar to the constraint that prevents *orphan* blocks in the seeding problem [40]. For evaluation purposes, we will also refer to a relaxed formulation of the problem without the load-balancing constraint. In that version, constraints 8, 9, 10, and 13 are removed, considerably reducing the problem complexity.

The ILP formulation given in this paper is designed for the most general case of data migration, where any file can be remapped to any volume. In reality, the migration goal might restrict some of the remapping options, potentially simplifying the ILP instance. For example, we can limit the set of volumes that files can be migrated to by eliminating the x_{fst} and c_{bst} variables where t is not in this set. We can similarly restrict the set of volumes files may be migrated from, or require that a set of specific files are (or are not) remapped.

Complexity and run time. The complexity of the ILP instance depends on $|B|$, $|F|$, and $|V|$—the number of blocks, files, and volumes, respectively. The number of variables is $|V|^2|F| + |V|^2|B| + |V| \times |B|$, corresponding to variable types x_{fst}, c_{bst}, and d_{bv}. Each of the constraints defined on these variables contributes a similar order of magnitude. An exception is constraint 13, which reformulates the system size, twice, to ensure that each individual volume's size is within the required margin. Indeed, the relaxed formulation without this constraint is significantly simpler than the full formulation.

We use two of the acceleration methods suggested by GoSeed to address the high complexity of the ILP problem. The first is *fingerprint sampling*, where the problem is solved for a subset of the original system's blocks. This subset (*sample*) is generated by preprocessing the block fingerprints and including only those that match a predefined pattern. Specifically, as suggested in [28], a sample generated with sampling degree k includes only blocks whose fingerprints consist of k leading zeroes, reducing the number of blocks in the problem formulation by $1/2^k$ on average.

The second acceleration method is *solver timeout*, which halts the ILP solver's execution after a predetermined runtime. As a result, the server returns a feasible solution that is not necessarily optimal. A feasible solution to the ILP problem can be directly translated into a migration plan, i.e., a list of files to migrate and their destination volumes. Thus, even if the solution is not optimal (due to sampling or timeout), the process still produces a valid plan for the original system.

We do not repeat the detailed analysis of the effectiveness of these heuristics, which were shown to be effective in earlier studies. Namely, the analysis of GoSeed showed that most of the solver's progress happens in the beginning of its execution (hence, timing out does not degrade its quality too much), and that it is more effective to reduce the sample size than to run the solver longer on a larger sample, as long as the sampling degree is not higher than $k = 13$. Our experiments with the extended ILP formulation, omitted due to space considerations, confirmed these findings.

6 Clustering

Overview. Clustering is a well-known technique for grouping objects based on their similarity [1]. 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 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 duplicates, 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.

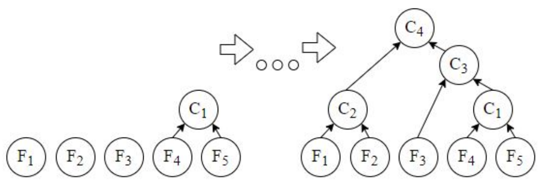
($H4$) **Multiple executions** Our heuristics introduce several parameters which we would be loath to overfit. We use the same initial state to perform repeated executions of the clustering process with multiple sets of parameter combinations (180 in our case), and choose the best migration plan from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

Hierarchical clustering. *Hierarchical clustering* [26] 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 3 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 3 represent the files in Figure 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 3, 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.

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* [26] for this purpose. For two sets A and B, their index is defined as $J(A,B) = {}^{|A \cap B|}/_{|A \cup B|}$. We view each file as a set



	F_1	F_2	F_3	F_4	F_5
F_1	-	-	-	-	-
F_2	2/3	-	-	-	-
F_3	1	4/5	-	-	-
F_4	2/3	2/3	4/5	-	-
F_5	1	3/4	3/4	1/3	-

$\xrightarrow{F4,F5}$

	F_1	F_2	F_3	C_1
F_1	-	-	-	-
F_2	2/3	-	-	-
F_3	1	4/5	-	-
C_1	1	3/4	4/5	-

$\xrightarrow{F1,F2}$

	C_2	F_3	C_1
C_2	-	-	-
F_3	1	-	-
C_1	1	4/5	-

$\xrightarrow{F3,C_1}$

	C_2	C_3
C_2	-	-
C_3	1	-

Figure 3: Hierarchical clustering with the files from Figure 1 (left) and the distance matrices created in the process (right).

of blocks, and thus, the Jaccard index for a pair of files is the portion of their shared blocks. 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 3 depicts the *distance matrix* for the files in Figure 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 recipes 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. Specifically, we use *complete linkage*, defined as follows: $dist_J(A \cup B, C) = max\{dist_J(A,C), dist_J(B,C)\}$. For example, the row for C_1 in the second distance matrix in Figure 3 lists the distances between C_1 and each of the remaining files.

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 3, $dist_V(C_1) = {}^1\!/_3$ 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. We address this limitation by multiple executions, described below.

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. 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 $\varepsilon = 5\%$ as our default.

Sensitivity to sample $(H3)$**.** As in the ILP-based approach, 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.

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 $\times\ (1 + gap)$.

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 section. 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.

7 Evaluation

We wish to answer two main questions: (1) how do the algorithms compare in terms of the final system size, load balancing, and runtime? and (2) how is the performance of the different algorithms affected by the various system and problem parameters? In the following, we describe our evaluation setup and the experiments executed to answer those questions.

7.1 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 [40]: The UBC dataset [7, 38] includes file systems of 857 Microsoft employees, of which we used the first 500 file systems (UBC-500). The FSL dataset [10] consists of snapshots of students' home directories at the FSL Lab at Stony Brook University [46, 47]. 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 [6]. 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 1. We emulate the ingestion of each snapshot

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 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.

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 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.

Implementation. All our algorithms are executed on a sample of the system's fingerprints, to reduce their 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 [40]: 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.

For our greedy algorithm (*Greedy*), we maintain a matrix where we record, for each block, the number of files pointing to it in each volume. We update this array to reflect the file remap performed in each iteration. To determine the space-saving ratio of each file, we reread its original snapshot file and lookup the counters of its blocks in the array. This is more efficient than keeping the list of each file's blocks in memory. Our Greedy implementation consists of 680 lines of C++ code.

For our ILP-based algorithm (*ILP*), we use the commercial Gurobi optimizer [3] as our ILP solver, and use its C++ interface to define our problem instances. We use a two-dimentional array similar to the one used for Greedy to calculate the set of blocks shared by each pair of volumes. We then create the variables and constraints as we process each snapshot file, freeing the original array from the memory before invoking the optimization by Gurobi. Our program for converting the input files into an ILP instance and retrieving the solution from Gurobi consists of 1860 lines of C++ code. We solve each ILP instance three times, each with a different random seed. The results in this section are the average of the three executions.

For our clustering algorithm (*Cluster*), we create a $|F| \times |B|$ bit matrix to indicate whether each file contains each block,

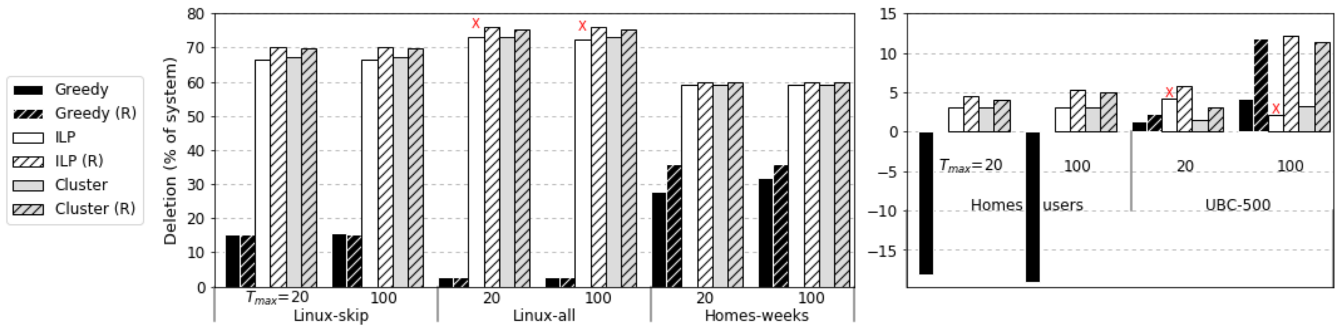


Figure 4: Reduction in system size of all systems and all algorithms (with and without load balancing constraints. $k = 13$ and $\mu = 2\%$).

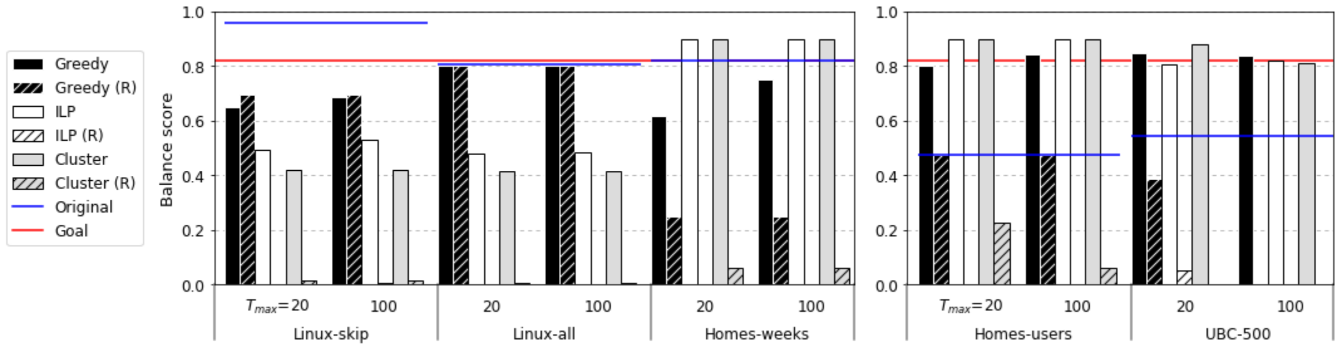


Figure 5: Resulting balance of all systems and all algorithms (with and without load balancing constraints. $k = 13$ and $\mu = 2\%$).

and use it to create the distance matrix (see Figure 3). 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 volumes. Our program consists of approximately 1000 lines of C++ code. 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.

Each algorithm has different resource requirements. Greedy is single threaded and requires a simple representation of the system's snapshot in memory. The ILP solver uses as much memory and as many threads as possible (38 in our case). The clustering algorithm ran in six processes, and used approximately 50% of our server's memory. We did not measure CPU utilization directly, although the runtime of the algorithms gives another indication of their compute overheads. Our implementation is open-source and available online [8]

7.2 Basic comparison between algorithms

Figure 4 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 the smallest deletion. For Homes-users, Greedy increased the system's size in attempt to comply with the load balancing constraint. 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 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 in our experiments) and returns a suboptimal solution. Specifically, the complexity of the UBC-500 system demonstrates an interesting limitation of ILP: its deletion with $T_{max} = 20\%$ is higher than with $T_{max} = 100\%$. The reason is that the solution space grows with T_{max}, and thus the best solution found when the solver times out is farther from the optimum.

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 5 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

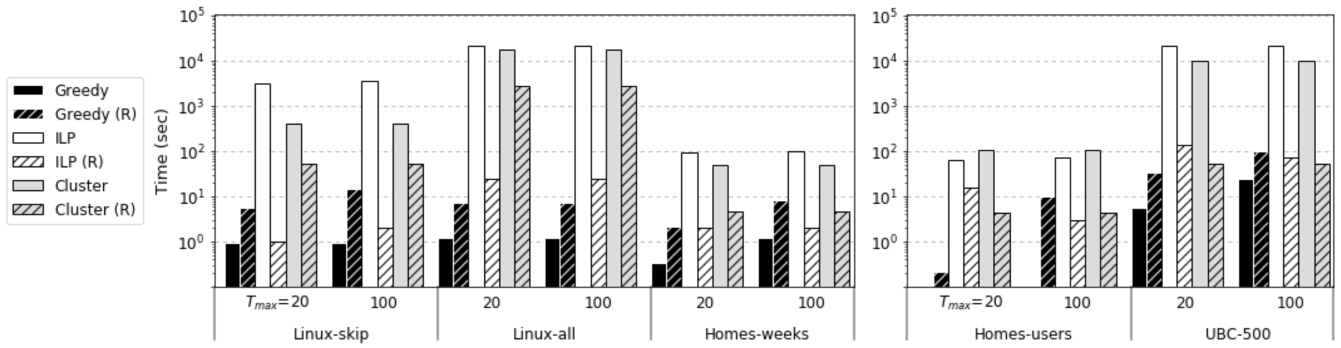


Figure 6: Algorithm runtime for all systems and all algorithms (with and without load balancing constraints. $k = 13$ and $\mu = 2\%$).

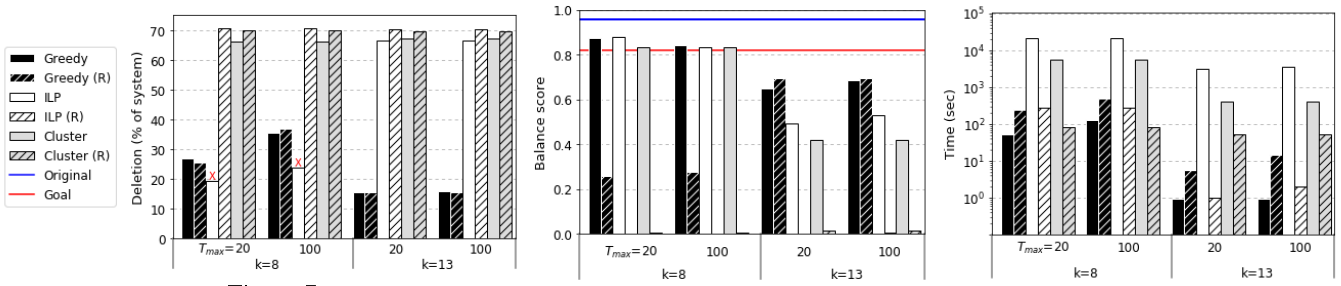


Figure 7: Linux-skip system with 5 volumes, $\mu = 2\%$, and two sampling degrees: $k = 8, 13$.

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 (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 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 in all our experiments. 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 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 [40] and space reclamation [41]. 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.

7.3 Sensitivity to problem parameters

Effect of sampling degree. Figure 7 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.

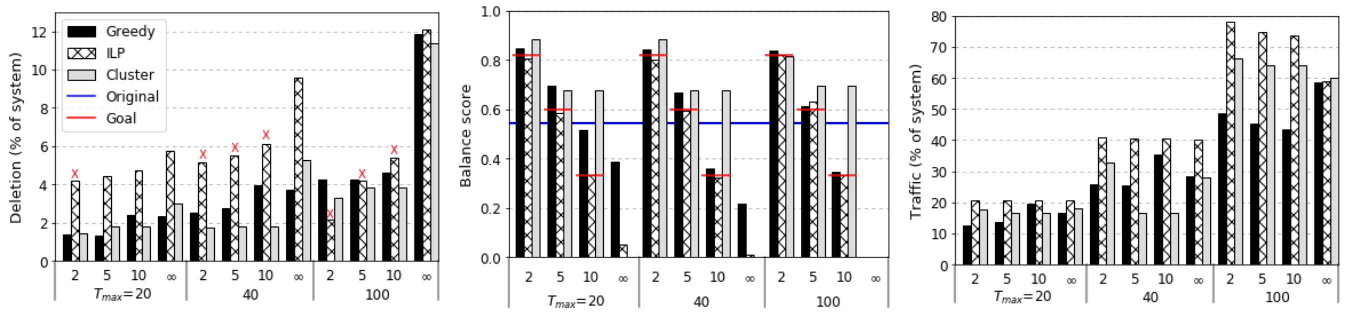


Figure 8: UBC-500 system with $k = 13$ and different load balancing margins.

The sample size affects each algorithm differently. Greedy achieves a higher deletion on the larger sample (by up to 238%), as it identifies more opportunities for capacity reduction. In contrast, ILP suffers from the increase in the problem size: it spends more time on finding a feasible solution and has less time for optimization, and thus its deletion on the larger sample is smaller. We repeated the execution of ILP on the large ($k = 8$) sample with a longer timeout—twelve hours instead of size—but the increase in deletion was minor. This confirmed the observation made for GoSeed [40], that it is more effective to reduce the sample size than to increase the runtime of the ILP solver. The relaxed ILP instance is much simpler, and thus relaxed ILP does not suffer such degradation. Cluster returns similar results for both sample sizes. The differences in the accuracy of the sample are masked by its randomized process.

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. We note that Greedy is so much faster than ILP and cluster, that its runtime on the large sample is considerably shorter than their runtime on the small one. Thus, if the sample is generated on-the-fly for the purpose of constructing the migration plan, it is possible to execute Greedy on a larger sample for a better migration plan.

Effect of load balancing and traffic constraints. Figure 8 shows the deletion, balance, and traffic consumption of all the algorithms on the UBC-500 system with different values of T_{max} and μ. The results on this system shows the highest sensitivity to these constraints due to the lower similarity between the 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 4. 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. Increasing the margin increases the runtime (not shown) of Greedy, as a result of more space-reduction iterations, as discussed above. The runtime of ILP and Cluster is not affected by the precise value of μ.

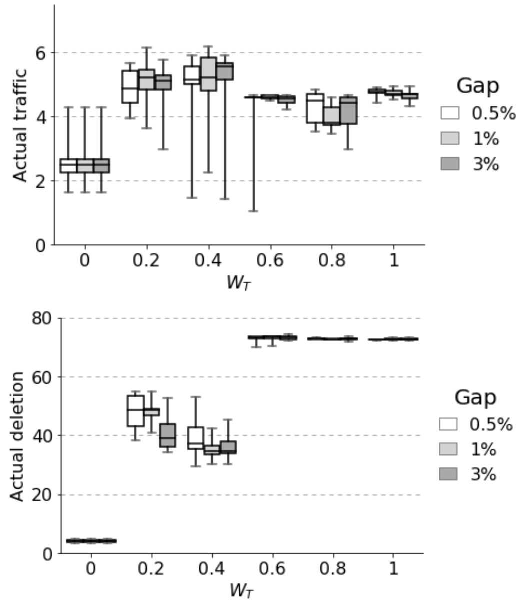


Figure 9: The distribution of migration traffic (top) and reduction in system size (bottom) in the set of plans returned by Cluster for Linux-all with $k = 13$.

Effect of randomization on Cluster. Figure 9 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. Recall that Cluster picks the plan with the highest deletion that complies with the traffic and load-balancing constraints.

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, which is why we repeat the clustering process for a range of values. Indeed, different W_T values result in very different deletions. For a given W_T, the range of deletion and traffic values generated by different *gaps* are similar. Thus, as no *gap* consistently 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.

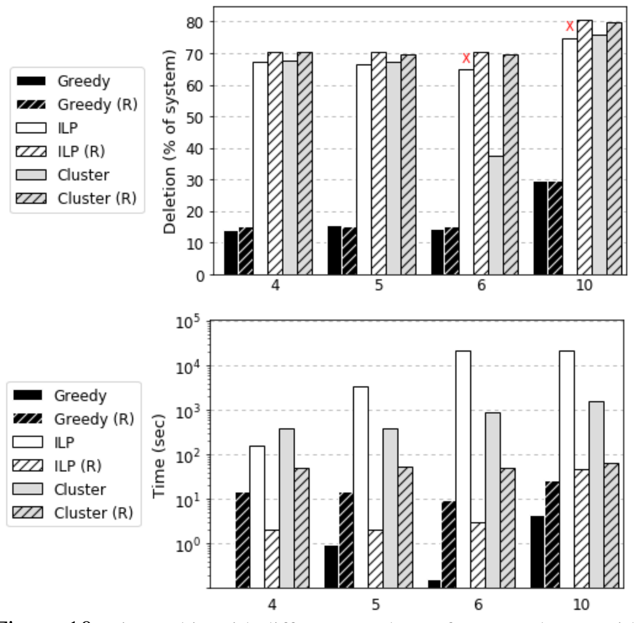


Figure 10: Linux-skip with different numbers of target volumes with $T_{max} = 100, k = 13, \mu = 2\%$.

We compared these results to final plans generated from 5 and 15 seeds (90 and 270 runs, respectively). The comparison, omitted due to space considerations, showed that using more than 5 random seeds carries diminishing returns. Thus, in practice, it is possible to halt the algorithm when additional runs do not improve the best solution so far.

Effect of the number of volumes. Figure 10 shows the deletion and runtime of our 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), because most of its traffic is spent on the faster load-balancing step. The runtime for a system with 10 volumes is much longer than for a system with only five volumes because there are more volume pairs and thus more file remap options to consider in each iteration. 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. Nevertheless, all our algorithms successfully generated migration plans for a varying number of volumes, most of them within less then an hour.

8 Conclusions and Future Challenges

We formulated the general migration problem for storage systems with deduplication, and presented three algorithms for generating an efficient migration plan. Our evaluation showed that the greedy approach is the fastest but least effective, and that the 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.

All our approaches 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 Greedy to generate multiple plans with different traffic constraints. These plans are points on the Pareto frontier [55], i.e., they represent different tradeoffs between the conflicting objectives of maximizing deletion and minimizing traffic. The multiple executions in the clustering algorithm provide a similar set of options.

Applying our approach in a live deduplicated system introduces several challenges, such as collecting and generating the system's snapshot as input to the algorithms, efficiently updating the metadata, determining the migration schedule, and adjusting it if new files are added to the system during this process. We leave these challenges for future work.

Acknowledgments

We thank the reviewers and our shepherd, Dalit Naor, for their feedback and suggestions. We thank Aviv Nachman for help with the ILP approach, Nadav Elias for the Linux snapshots, and Danny Harnik for insightful discussions. This research was supported by the Israel Science Foundation (grant No. 807/20).

References

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

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

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

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

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

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

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

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

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

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

[11] 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.

[12] 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.

[13] 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.

[14] 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.

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

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] Min Fu, Dan Feng, Yu Hua, Xubin He, Zuoning Chen, Wen Xia, Fangting Huang, and Qing Liu. Accelerating restore and garbage collection in deduplication-based backup systems via exploiting historical information. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, 2014.

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

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

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

[28] 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.

[29] 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.

[30] Michal Kaczmarczyk, Marcin Barczynski, Wojciech Kilian, and Cezary Dubnicki. Reducing impact of data fragmentation caused by in-line deduplication. In *Proceedings of the 5th Annual International Systems and Storage Conference (SYSTOR 12)*, 2012.

[31] 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.

[32] Mark Lillibridge, Kave Eshghi, and Deepavali Bhagwat. Improving restore speed for backup systems that use inline chunk-based deduplication. In *11th USENIX Conference on File and Storage Technologies (FAST 13)*, 2013.

[33] 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.

[34] 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.

[35] 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.

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

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

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

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

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

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

[42] 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.

[43] 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.

[44] 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.

[45] 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.

[46] 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.

[47] 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.

[48] 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.

[49] 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.

[50] 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.

[51] 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.

[52] 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.

[53] zhichao Cao, Hao Wen, Fenggang Wu, and David H.C. Du. ALACC: Accelerating restore performance of data deduplication systems using adaptive look-ahead window assisted chunk caching. In *16th USENIX Conference on File and Storage Technologies (FAST 18)*, 2018.

[54] 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.

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

Appendix

Formal formulation of constraints and objective. The ILP formulation for migration with load balancing consists of 13 constraint types.

1. All Variables are Boolean: $x_{fst}, c_{bst}, d_{bv} \in \{0,1\}$
2. A file can be remapped to at most one volume: for every file f in volume s, $\sum_{t\in V} x_{fst} \leq 1$.
3. A block can only be deleted or copied from a volume it was originally stored in: for every two volumes s,t; if $b \notin s$ then $c_{bst} = d_{bs} = 0$.
4. A block can be deleted from a volume only if all the files containing it are remapped to other volumes: for every volume s and for every file f such that $f \in s$, $d_{bs} \leq \sum_t x_{fst}$.
5. A block can be deleted from a volume only if no file containing it is remapped to this volume: for every two volumes s,t, every file f such that $f \in s$ and $f \notin t$, and every block b such that $(b,f,s) \in I_S$, $d_{bt} \leq 1 - x_{fst}$.
6. View all the blocks in the volume intersections as having been copied: for every two volumes s,t and for every block $b \in Intersect_{st}$, $c_{ist} = 1$.
7. When a file is remapped, all its blocks are either copied to the target volume, or are initially there (as part of the intersection): for every two volumes s,t and every block b and file f such as $(b,f,s) \in I_S$, $x_{fst} \leq \Sigma_{v\in V} c_{bvt}$.
8. A block can be copied to a target volume only from one source volume: for every block b and volume t, $\Sigma_{s \text{ such that } b\notin Intersect_{st}} c_{bst} \leq 1$.
9. A block must be deleted if there are no files containing it on the volume: for every two volumes s,v and all files $f_s \in s$, $f_v \in v$ and all blocks b where $b \in f_s$ and $b \in f_v$, $d_{bs} \geq 1 - \{\Sigma_{f_s}(1 - \Sigma_v x_{f_s sv}) + \Sigma_{fv}(x_{f_v vs})\}$.
10. *A* block cannot be copied to a target volume if no file will contain it there: For every volume t and every block $b \notin t$, $\Sigma_s c_{bst} \leq \Sigma_s \Sigma_{f\in s \wedge b\in f} x_{fst}$
11. A file cannot be migrated to its initial volume: for every file f and volume v, $x_{fvv} = 0$
12. *Traffic constraint*: the size of all the copied blocks is not larger than the maximum allowed traffic: $\sum_{s\in V}\sum_{t\in V}\sum_{b\notin Intersect_{st}} c_{bst} \times size(b) \leq T_{max}$.
13. *Load balancing constraint*: for each volume v, $(w_v - \mu) \times Size(S') \leq Size(v') \leq (w_v + \mu) \times Size(S')$, where $Size(v')$ is the volume size after migration, i.e., the sum of its non-deleted blocks and blocks copied to it: $Size(v') = \sum_{b\in v}(1 - d_{bv}) \times Size(b) + \sum_{s\in V, \sum_b \notin Intersect_{sv}} c_{bsv} \times Size(b)$. $Size(S')$ is the size of the system after migration: $Size(S') = \sum_{v\in V} Size(v')$.

► *Objective*: maximize the sum of sizes of all blocks that are deleted minus all blocks that are copied. This is equivalent to minimizing the overall system size: $Max\left(\sum_{b\in B} Size(b) \times \sum_{s\in V}\left[d_{bs} - \sum_{t\in V, b\notin Intersect_{st}} c_{bst}\right]\right)$.