Algorithmic Problems in Data Management

Samir Khuller
Dept. of Computer Science
University of Maryland
College Park, MD 20742
samir@cs.umd.edu
(301) 405 6765
URL:http://www.cs.umd.edu/∼samir/

May 22, 2010

Contents

1 Introduction

The primary goal of this paper is to highlight some interesting research problems arising in the context of data management issues in large scale storage systems. Large scale storage systems are becoming increasingly prevalent with data being collected via transactions, satellites, sensor networks etc. The volume of data that needs to be processed is easily in the Terabyte scale for even small size applications, and growing even more rapidly. Hence, they are critical components of distributed systems. Our objective is to address some fundamental problems dealing with storage issues that are encountered in such situations.

Centralized large scale storage systems are typically composed of a collection of disks connected on a very fast local area network (such a system is often also referred to as a *Storage Area Network (SAN)*). Such systems are used for storage applications, such as a database of videos, multimedia, or images. A SAN essentially allows multiple processors to access several storage devices. They typically access the storage medium as though it were one large shared repository. One crucial function of such a storage system is that of deciding the placement of data within the system. This data placement is dependent on the demand pattern for the data. For instance, if a particular data item is very popular the storage system might want to host it on a disk with high bandwidth or make multiple copies of the item. The storage system needs to be capable of handling flash crowds [9]. During events triggered by such flash crowds, the demand distribution becomes highly skewed and different from the normal demand distribution.

Each individual disk is primarily characterized by a few key parameters: namely the size of the disk, and its load capacity. In applications, where multimedia data is continuously being streamed to the client, the system cannot handle a very large number of clients. The total amount of bandwidth that each disk is able to sustain is limited, and this constrains the number of users that can access data stored on a single disk. For fault tolerance, higher throughput, and lower response time requirements, we may need to replicate objects since a single disk cannot handle extremely high demand. In addition, the precise layout is very important since the layout controls the efficiency of the overall system. For example, putting several popular objects on the same disk may cause the disk to fail when the demand spikes, or cause unacceptable delays for some users. We note, that for the remainder of this proposal, unless otherwise stated, by layout, we mean which object should be placed on which disk, rather than the physical layout of objects within a particular disk. Similar assumptions are made in the work of M. Henzinger (SODA 2007, invited plenary speaker) for the problem of load balancing on index servers for a search engine. As queries arrive, each query is mapped to a subset of devices that contain information relevant to the query. The results returned from each device are then sent to a processor for appropriate processing. The goal is to keep the maximum load low by finding a good layout, given the keywords access pattern and an index corresponding to each keyword.

Another aspect is that of locating data close to where the demand is. In much of our previous work, we assumed that the assignment cost of clients is uniform. In other words, the distance from the client to all disks is the same (for example if the collection of disks is located at the same place). Some approximation algorithms have been developed for the case when there is an assignment cost for clients to access data from a disk, which depends on the location of the disk. However, these algorithms either ignore the load capacity [4, 17], or violate storage and load capacities by a constant factor [7].

We note that a closely related effort to ours is the work at the University of Washington, done by Anna Karlin's group [2, 8]. However, one key difference is that they focused on the data migration problem by *fixing* the source and destination disks for the transfers to be performed by creating a "transfer graph". The algorithm now needs to work with graph coloring methods to find a good schedule. However, they extended these methods to deal with space constraints as well. The main drawback of this approach is that one may not find very good schedules since one does not optimize over the choice of transfer graphs (different transfer graphs give rise to schedules of widely varying costs [12]).

In summary, some of our *key goals* for centralized storage systems are as follows:

- How can we create a good initial layout on a collection of disks?
- How should we migrate data to adjust to changing demand? Moreover, we expect the system to automatically decide when the demand has changed sufficiently to re-organize its layout.

- How do we incorporate assignment costs into the solution? In other words, how do we keep data close to the demand?

2 Data Layout Issues

We first describe some of the progress we have made over the last few years in addressing these questions. The ideas and methods will be introduced by toy examples, but we hope this will illustrate the point we are trying to make. The reader is urged to read the full papers to understand the detailed algorithms and proofs [5, 10, 12, 6, 11, 13].

We have been studying such storage systems for several years. The first fundamental problem we considered was the following. We are given a collection of M data objects, and a collection of N identical disks, with each disk having storage capacity k and load capacity L. For each data object i, we know the demand d_i for this object. The goal is to decide (a) for each data object how many copies should we create and (b) which subset of disks should we place this object on. This is referred to as the *data layout* problem.

A simple example follows: suppose we have two disks, each with load capacity 100 and storage capacity 2. In other words, $N = 2, k = 2, L = 100$. We will assume that $M = 4$ (notice that Nk is an upper bound on the total number of data objects that can be stored in the system). We have 4 data objects A, B, C, D with demands $d_A = 120, d_B = 30, d_C = 10, d_D = 40$. Notice that there is no "perfect" layout. In other words one option would be to put A and C on one disk, and B and D on the second disk. Notice that the load on the first disk will be 100, but the load on the second disk will be only 70. 30 customers will not be able to access the data that they need. So one assignment of the load could be: 90 units of demand for A, and 10 units of demand for C are assigned to the first disk, and all the demand for B and D is assigned to the second disk. However, a better layout will be to put A and B (this satisfies 90 units of demand, 60 for A and 30 for B) on one disk and A and D on another disk (this satisfies 100 units of demand, 60 for A and 40 for D) and drop C altogether if we only wish to maximize total satisfied demand with no consideration for fairness. In any case, this already shows that the layout can crucially affect the effectiveness of a system to cope with non-uniform demand for data.

The sliding-window algorithm (SW) developed by Shachnai and Tamir [18] was proposed for the data layout problem. In SODA 2000, we showed [5] that this algorithm is in fact an extremely good algorithm for the problem. In particular, if we assume that $M \leq Nk$ and $\sum_i d_i \leq NL$, then we showed that *regardless* of the demand distribution for the data objects, the sliding window algorithm computes a layout in which at least $(1 - \frac{1}{(1+\sqrt{k})^2})$ fraction of the demand can always be satisfied[1]. Moreover, we showed that this bound is tight! In other words, there are inputs for which there is no layout that can satisfy a larger demand. Note that this function rapidly goes to 1 as k increases, and is at least 0.75 even when $k = 1$. We also proved that the problem of finding an optimal solution is NP-hard even for identical disks. However we were able to show experimentally, that this algorithm is very fast and produces almost optimal solutions. In fact, we also established an $O((N+M)\log(N+M))$ worst case bound on the running time of the algorithm, improving the previous bound of $O(NM)$ given in [18]. In addition we developed a polynomial time approximation scheme, but this algorithm is not practical.

Subsequently, we were able to extend some of these results for the case where the data objects themselves may have different sizes chosen from a set $\{1, \ldots, \Delta\}$. In this case, we developed a new algorithm [10] and showed that the fraction of demand that can always be satisfied is $\frac{k-\Delta}{k+\Delta}\left(1 - \frac{1}{\left(1+\sqrt{\frac{k}{2\Delta}}\right)^2}\right)$; but this bound is not tight. However, as with the sliding window algorithm for the unit size object case, the algorithm is practical and delivers solutions of very good quality. For the simple case when $\Delta = 2$ we were able to obtain a tight bound of $(1 - \frac{1}{(1+\sqrt{\lfloor k/2 \rfloor})^2})$ by using a more complex algorithm. However, the proof and algorithms are complex, and this approach is not easy to extend to the case when Δ is arbitrary.

One important open problem remaining is to develop a tight bound for the data layout problem for arbitrary sized

[1] In the simple example given above, note that 190 clients out of 200 are satisfied. This means that 0.95 fraction of the clients are satisfied. In fact this layout is precisely the one computed by the SW algorithm.

objects chosen from the set $\{1, \ldots, \Delta\}$ *and uniform disks. Our conjecture is that the bound will be* $(1 - \frac{1}{(1+\sqrt{\lfloor k/\Delta \rfloor})^2})$. We have examples showing that there are layouts for which we cannot do any better; it would be fantastic if we can close the gap and show that this is indeed the worst possible example.

Another important consideration is that storage systems are built incrementally over time, and this gives rise to heterogeneous systems and not homogeneous systems. We were able to extend some of the results to the case of heterogeneous systems [5], when the ratio of load to storage capacity remained (roughly) uniform. This is reasonable since as technology improves, disks get larger *and* faster. However, one may have different options when building a system which include a tradeoff in storage and speed. It would be useful to develop algorithms that can work with an arbitrary collection of disks without making any assumptions about their parameters. In this case, it is easy to generate examples that show that we cannot prove a bound by comparing to the *total* demand since even the optimal solution cannot satisfy most of the demand. However, we may be able to prove a bound by comparing to the optimal solution, rather than the total demand.

Finally, the most general problem we would like to solve is to develop a layout algorithm for *arbitrary sized objects* on a heterogeneous disk system. Clearly, this problem is related to classical bin packing and multiple knapsack type problems which are also NP-hard. In addition, clients may have demands for certain objects, and there is a cost function c_{ij} that specifies the cost for client i to access disk j. Our goal is to find a layout where a specified amount of demand is satisfied at minimum cost. This formulation is reasonable, since there may be no solution that satisfies all the demand (as argued earlier). In this case, we would like to satisfy a particular fraction of the total demand, with minimum assignment cost. At the same time we should not violate the load and storage capacities of the disks. Another interesting version is when we actually require all the demands to be satisfied, and would like to minimize the assignment cost and are allowed to violate the storage capacities by an additive unit factor. Without assignment costs, we can always satisfy all the demand by violating the storage requirements by an additive unit factor. This is a simple corollary of our proof [5] and the main result in [18].

Previous work takes into account storage constraints and assignment costs, but ignores load capacities [4] (further improved bounds approximation factors have been obtained by Swamy (unpublished paper)). Their approach is based on LP-rounding, and they obtain constant factor approximation algorithms. A constant factor bi-criteria approximation is given in [7], where load and storage constraints are allowed to be violated. We would like to consider a local search based approach as has been successfully used both for the K-median problem, and the facility location problem [3].

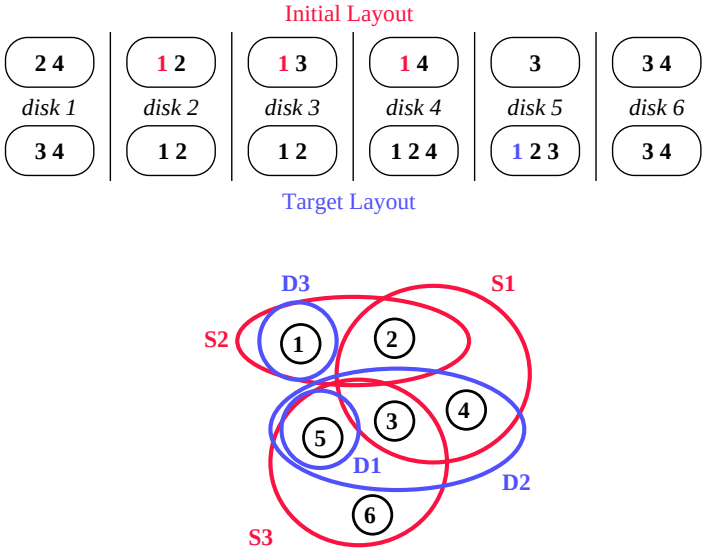
3 Dynamically Changing Demand

Over time, the demand for data may change dynamically. Earlier [12] we argued that demand estimates may be used to compute an initial layout. However, over time as the demand patterns change, the system may wish to reconfigure itself to respond to changing demand. Dealing with dynamically changing demand gives rise to very interesting and important challenges.

(a) We assume that the original layout was computed based on an "estimate" of the demand distribution for data. However, over time we might realize that this "estimate" was not very accurate and wish to change it. How/when should we do that? How frequently should this estimate be updated?

(b) The second question is about re-arranging the layout to cope with the new demand pattern. Clearly, doing this quickly is important since the system is running inefficiently as long as the desired layout has not been reached. In addition, performing the migration itself uses some of the bandwidth that is allocated for use by clients.

We expect that the system will continue to function (handling client traffic) while the data re-organization is being done in the background. A certain amount of bandwidth will be dedicated for background (migration) traffic. Our initial approach was the following [12]: we used the Sliding Window algorithm to recompute a new "target" layout $L_{\mathcal{T}}$ from



the demand distribution $\mathcal{T}$. Once we computed the new target layout $L_{\mathcal{T}}$, our objective was to "convert" the existing layout to the new layout. The conversion process can be described formally as follows: we have a collection of N disks, and for each item i we have a set of "source" disks S_i and a set of "destination disks" D_i (these are the disks that need item i, but do not have it as yet). In the figure shown, for example item 1 is initially on disks 2,3 and 4. Hence $S_1 = \{2, 3, 4\}$. The item has to be sent to disk 5, which does not have it. Hence $D_1 = \{5\}$. The conversion proceeds in a series of "rounds". In each round, we can choose a pairing of disks (a matching). For each chosen pair, one disk can transfer one data item to the one it is paired with. In other words, each disk can either be the sender or recipient of a data item. The goal is to minimize the number of rounds. In [12] we showed that this problem is NP-hard, and we developed a constant factor approximation algorithm for it [12, 15]. We also implemented this algorithm and compared its performance to several natural heuristics [6]. In fact, both the approximation algorithm and the heuristics are able to deliver solutions that are very close to the optimal most of the time. In addition note that the total number of transfers performed here is *optimal*, since each transfer done is a required transfer since we only transfer data from a disk in source set to a disk in the corresponding destination set. Recently we have been able to improve the approximation factor of the algorithm from 9.5 to 6.5 by using a combination of new ideas [15]. In addition we were able to obtain a better approximation factor of 4 using the full-duplex communication model, where each disk can send and receive one item in each round [15].

Different communication models can be considered based on how the disks are connected. We use the same model as in [2, 8] where the disks may communicate on any matching; in other words, the underlying communication graph allows for communication between any pair of devices via a matching (e.g., as in a switched storage network with unbounded backplane bandwidth). *This model best captures an architecture of parallel storage devices that are connected on a switched network with sufficient bandwidth. This is most appropriate for our application*. This model is one of the most widely used in all the work related to gossiping and broadcasting. These algorithms can also be extended to models where the size of the matching in each round is constrained [12]. This can be done by a simple simulation, where we only choose a maximal subset of transfers to perform in each round. We also consider a full duplex model[15] in which each disk can send and receive an item at the same time in a round.

The main problem with the above approach is that it completely *ignores the current storage pattern*. In other words, when we run the SW algorithm to compute a new layout, it completely ignores the current layout. Since there are many possible layouts that are potentially similar in their performance, perhaps we could have chosen a "target" layout that is a good one, and at the same time easy to obtain by modifying the existing layout. In fact, this turns out to be an extremely important point. In recent work [11] we have been able to show experimentally that this is indeed a viable option. For example, in tests where the previous approach took over 100 rounds for the conversion of one layout to another; we were able to show that within 5-10 rounds, other layouts can be obtain that are almost as good as the layouts obtained

after over 100 rounds (See Fig.7). (Even though we were employing an approximation algorithm, we were able to derive lower bounds that were very close to the upper bounds. Hence these were not off by large factors.) However, we are (as yet) unable to prove bounds of the form: given any existing layout, a target bound of δ rounds, one can convert the existing layout to a new layout within δ migration rounds and always satisfy f-fraction of the demand. Clearly, if δ is allowed to be very large then $f = (1 - \frac{1}{(1+\sqrt{k})^2})$. This would be a very nice result to show as it would completely explain the tradeoff between the number of migration rounds and quality of solution. In the previous approach, we essentially fixed $f = (1 - \frac{1}{(1+\sqrt{k})^2})$ and then tried to minimize δ. Since finding the optimal solution was NP-hard, our objective was to approximate δ.

We now consider a new approach to deal with the problem of changes in the demand pattern. We ask the following question:

In a given number of migration rounds, can we obtain a layout by making changes to the existing layout so that the resulting layout $\mathcal{L}'$ will be (close to) the best possible layout that we can obtain within the specified number of rounds?

Of course, such a layout is interesting only if it is significantly better than the existing layout for the new demand pattern. A simpler question may be to find a layout which has the property that each disk has at most one new data item. While there are cases that show that such a change might not be achievable in a constant number of rounds of matching, studying this problem would yield some insights for the main question we are interested in.

We approach the problem of finding a good layout that can be obtained in a specified number of rounds by trying to find a sequence of layouts. Each layout in the sequence can be transformed to the next layout in the sequence by applying a small set of changes to the current layout. These changes are computed so that they can be applied within one round of migration (a disk may be involved in at most one transfer per round).

We show that by making these changes even for a small number of consecutive rounds, the existing placement that was computed for the old demand pattern can be transformed into one that is almost as good as the best layout for the new demand pattern.

To do this, we define the following *one round problem*. Given a layout $L_{\mathcal{P}}$ and a demand distribution $\mathcal{T}$, our goal is to find a one round migration (a matching), such that if we transfer data along this matching, we will get the maximum increase in utilization. In other words, we will "convert" the layout $L_{\mathcal{P}}$ to a new layout $L_{\mathcal{P}+1}$, such that we get the maximum utilization, and the new layout is obtainable from the current layout in one round of migration.

Now we can simply use an algorithm for the *one round problem* repeatedly by starting with the initial layout $L_{\mathcal{I}}$, and running ℓ iterations of the one round algorithm. We will obtain a layout $L_{\mathcal{I}+\ell}$, which could be almost as good is the target layout $L_{\mathcal{T}}$.

Of course there is no reason to assume that repeatedly solving the one round problem will actually yield an optimal solution for the ℓ round version of this problem. However, as we will see, this approach is very effective. (In fact, the one round migration problem is also NP-hard as was shown in [11], and we have to resort to using a heuristic for it.)

One important additional constraint is that the transfers have to be done without violating the space constraints on a disk. In our current approach we require the space constraints to be satisfied at all times. In previous work we ignored the temporary space requirements. Space requirements were addressed in related work by Karlin's group [2, 8], however their formulation of the problem focuses on how to schedule migrations for a specified set of transfers (where a decision has already been made to migrate an object from a specified source disk to a target disk). With the assumption that each disk has only one spare unit of storage, they are able to schedule all the move operations specified by the transfer graph.

Example Since the formal definition of the problem involves a lot of notation, we will informally illustrate the problem and our approach using an example. In this example, we will show an initial demand distribution $\mathcal{I}$; an initial placement for this distribution $L_{\mathcal{I}}$; we will then show the changed demand distribution $\mathcal{T}$. We will show why the initial placement $L_{\mathcal{I}}$ is inadequate to handle the changed demand distribution $\mathcal{T}$. We will then show how a small change (a one-round migration) to the initial placement $L_{\mathcal{I}}$ results in a placement that is optimal for the new demand distribution.

In this toy example, we consider a storage system that consists of 4 identical disks. Each disk has storage capacity of 3 units and load capacity (or bandwidth) of 100 units. There are 9 data items that need to be stored in the system. The

initial demand distribution $\mathcal{I}$ and the new demand distribution $\mathcal{T}$ are as follows:

Item	Initial demand	New demand
A	130	55
B	90	55
C	40	20
D	30	60
E	25	5
F	25	10
G	25	15
H	22	70
I	13	110

The placement $L_{\mathcal{I}}$ (which in this case is also an optimal placement) obtained using the sliding window algorithm[2] for the demand distribution above is as follows (the numbers next to the items on disks indicates the mapping of demand to that copy of the item):

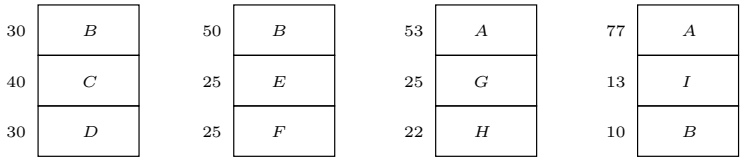


Figure 1: Optimal placement $L_{\mathcal{I}}$ for the initial demand distribution $\mathcal{I}$, satisfies all the demand. Storage capacity k=3, Bandwidth L=100. In addition to producing the layout the sliding window algorithm finds a mapping of demand to disks, which is optimal for the layout computed.

To determine the maximum amount of demand that the current placement $L_{\mathcal{I}}$ can satisfy for the new demand distribution $\mathcal{T}$, we compute the max-flow in a network constructed as follows. In this network we have a node corresponding to each item and a node corresponding to each disk. We also have a source and a sink vertex. We have edges from item vertices to disk vertices if in the placement $L_{\mathcal{I}}$, that item was put on the corresponding disk. Capacities of edges from the source to every item is equal to the demand for that item in the new distribution. The rest of the edges have capacity equal to the disk bandwidth. Using the flow network above, we can re-assign the demand $\mathcal{T}$ using the same placement $L_{\mathcal{I}}$ as given in Figure 3. Figure 2 shows the flow network obtained by applying the construction described above, corresponding to the initial placement $L_{\mathcal{I}}$ and new demand $\mathcal{T}$. (For the initial demand, the max flow had value 400.)

A small change can convert $L_{\mathcal{I}}$ to an optimal placement. In general, we would like to find changes that can be applied to the existing placement in a single round to obtain a placement that is close to an optimal placement for the new demand distribution. In a round a disk can either be the source or the target of a data transfer but not both. In fact, in this example a single change that involves copying an item from one disk to another is sufficient (and does not involve the other two disks in data transfers). This is illustrated in Figure 4.

We stress that we are not trying to minimize the total number of data transfers, but simply find the *best* set of changes that can be applied in parallel to modify the existing placement for the new demand distribution.

We compare this approach to that of previous works [12, 6] which completely disregard the existing placement and simply try to minimize the number of parallel rounds needed to convert the existing placement to an optimal placement for the new demand distribution. In Fig. 6, we show that using the old approach, it takes 4 rounds of transfers to achieve what our approach did in a single round (and using just one transfer). In Figure 5 an optimal placement $L_{\mathcal{T}}$ is recomputed [3] for the new demand distribution $\mathcal{T}$. We show in Figure 6 the smallest set of transfers required to convert

[2]The sliding window algorithm proposed by Shachnai and Tamir [18] is currently the best practical algorithm for this problem. For more on the sliding window algorithm and its performance, see [5].

[3]Using the sliding window algorithm for computing a placement for a given demand.

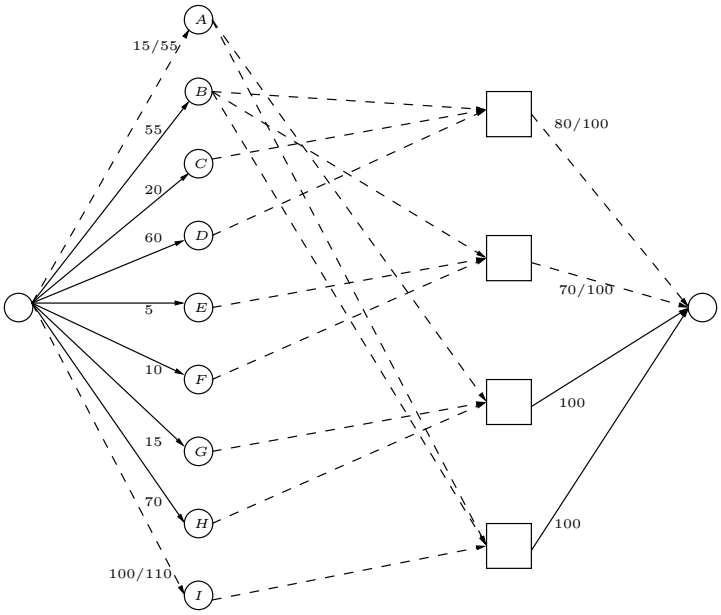


Figure 2: Flow network to determine maximum benefit of using placement $L_{\mathcal{I}}$ with demand distribution $\mathcal{T}$. $L_{\mathcal{I}}$ is sub-optimal for $\mathcal{T}$ and can only satisfy 350 out of a maximum of 400 units of demand. Saturated edges are shown using solid lines. f/c denotes the flow and capacity of an edge.

0	B	55	B	15	A	0	A
20	C	5	E	15	G	100	I
60	D	10	F	70	H	0	B

Figure 3: Maximum demand that placement $L_{\mathcal{I}}$ can satisfy for the new demand distribution $\mathcal{T}$. $L_{\mathcal{I}}$ is sub-optimal for $\mathcal{T}$ and can only satisfy 350 out of a maximum of 400 units of demand.

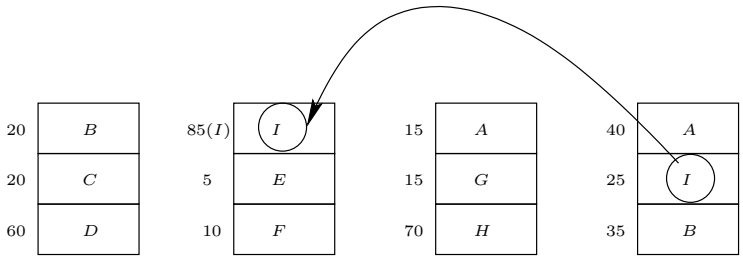


Figure 4: Removing item B from disk 2 and replacing it with a copy of item I from disk 4 converts $L_{\mathcal{I}}$ to an optimal placement $\mathcal{L}'$ for the new demand distribution $\mathcal{T}$. The placement shown above is optimal for $\mathcal{T}$ and satisfies all demand.

$L_{\mathcal{I}}$ to $L_{\mathcal{T}}$. *Note that both placement $\mathcal{L}'$ (obtained after the transfer shown in Figure 4 is applied) and placement $L_{\mathcal{T}}$ shown in Figure 5 are optimal placements for the new demand distribution $\mathcal{T}$.* Note that this is an optimal solution that also addresses the space constraint on the disk (this property is not actually maintained by the data migration algorithms developed earlier [12]).

The heuristic that we developed is based on a fairly simple idea. For any disk d, let $I(d)$ denote the items on that disk. Corresponding to any placement $\{\mathbf{p_i}\}$ (a placement specifies for each item, which set of disks it is stored on), we

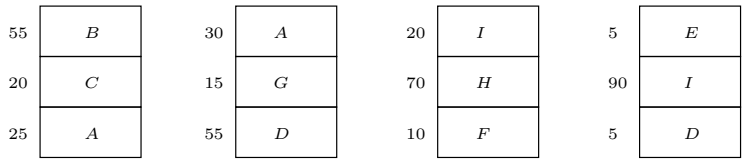


Figure 5: Placement $L_{\mathcal{T}}$. Output of the Sliding window algorithm for the new demand distribution $\mathcal{T}$.

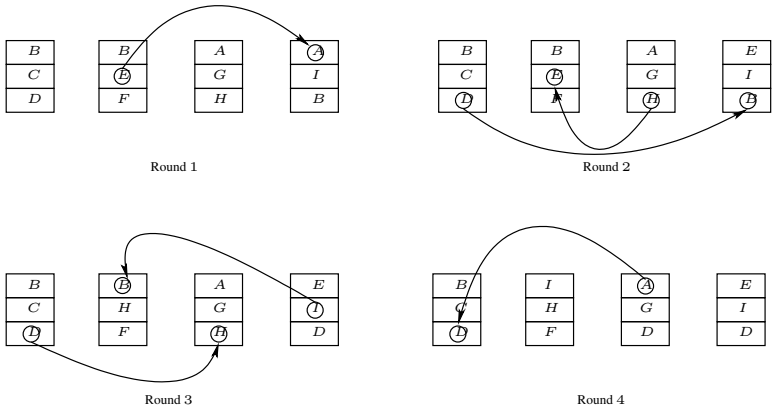


Figure 6: Transforming $L_{\mathcal{I}}$ to $L_{\mathcal{T}}$ takes 4 rounds. Note that the disks here will need to be renumbered to match the sliding window output. Final disk 2 corresponds to disk 3 in the sliding window output, final disk 3 corresponds to disk 2 in the sliding window output.

define the corresponding flow graph $G_p(V, E)$ as follows. We add one node a_i to the graph for each item $i \in \{1 \ldots \Delta\}$. We add one node d_j for each disk $j \in \{1 \ldots N\}$. We add one source vertex s and one sink vertex t. We add edges (s, a_i) for each item i. Each of these edges have capacity $\texttt{demand}(i)$ (where $\texttt{demand}(i)$ is the demand for item i). We also add edges (d_j, t) for each disk j. These edges have capacity L (where L is the load capacity of disk j). For every disk j and for every item $i \in I(j)$, we add an edge (a_i, d_j) with capacity L.

The algorithm starts with the initial placement and works in phases. At the end of each phase, it outputs a pair of disks and a transfer corresponding to that disk pair. We consider all possible transfers between each pair of disks, and choose the one that would maximize the flow when we change the flow graph constructed as described. We then greedily pick the best choice. That fixes one edge in the matching. We then repeat this to construct the entire matching, one edge at a time.

We can speed up the algorithm by observing that the max-flow value increases monotonically from one phase to the next and therefore we need not recompute max-flow from scratch for each phase. Rather, we compute the residual network for the flow graph once and then make incremental changes to this residual network for each max-flow computation. All max-flow computations in this version of the algorithm are computed using the Edmonds-Karp algorithm (see [1]). Let G_i denote the residual graph at the end of phase i. Let G_0 be the residual graph corresponding to the initial graph. All max-flow computations in phase $i+1$, we begin with the residual graph G_i and find augmenting paths (using BFS on the residual graph) to evaluate the max-flow. After each transfer pair in phase $i+1$ is considered, we undo the changes to the residual graph and revert back to G_i. At the end of phase $i+1$, we apply the best transfer found in that phase, recompute max-flow and use the corresponding residual graph as G_{i+1}.

Even with the speedup, the algorithm needs to perform around 415,000 max-flow computations even for one of the smallest instances (N=60, k=15) that we consider in our experiments. Since we want to quickly compute the one-round migration, too many flow computations are not acceptable. We therefore consider variants of our algorithm. In our experiments, we found these variants to yield solutions that are as good as the algorithm described above. Moreover these variants run in seconds, as opposed to hours for the brute force algorithm.

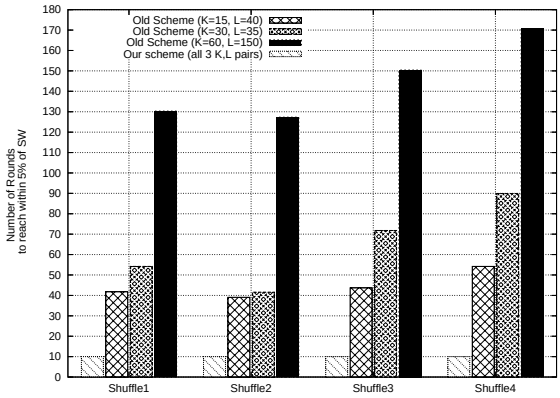


Figure 7: Plot compares the number of rounds that the old migration scheme took to reach within 5% of the optimal solution. We used N=60 and tried each of the shuffle methods for every pair of k and L shown in the plot. Every data point was obtained by averaging over 10 runs. In each of the experiments shown above, our scheme was set to run for 10 consecutive rounds.

We also are aware of situations when this heuristic does not perform well [11]. In fact, sometimes moving items whose load did not even increase can enable one to "merge" all the demand together on a disk freeing up lots of load capacity on several disks. To derive a constant factor approximation one will have to understand this phenomenon a little better. Another obvious approach would be to try to construct some graph where we have a vertex corresponding to each disk, and the edge weights encode the "benefit" of copying an item from one disk to another. We could then try to identify a maximum weight matching in this graph. However, the *main problem* with this approach is that the same set of clients can cause many edges in the maximum weight matching to be chosen, but all these transfers cannot give a benefit since they are all referring to the same set of clients.

4 Central Questions

1. One of our first goals is to develop a tradeoff between migration rounds and quality of layout. We think that this will greatly enhance our understanding of the problem. Even though we have developed a very effective heuristic, it would be nice to develop algorithms for which we can prove worst case bounds. In addition, we would like to improve our heuristic in terms of making it computationally more efficient. If we expect the algorithm to scale to larger storage systems, then we need faster algorithms than the one we presently have.

2. We would also like to develop tight bounds for the data layout problem with arbitrary sized items and *extend the migration results for the case of arbitrary sized items*. This immediately implies that we will have to use different communication models, as all transfers will not take the same amount of time, even on a homogeneous centralized system.

3. Several of our layout algorithms (with one notable exception) currently assume that the storage system consists of identical disks. We would like to develop algorithms that work for arbitrary disk systems.

4. For the data migration work we would like to extend our results for other communication models, in addition to the half-duplex model. The full duplex model permits each disk to send and receive data simultaneously for example. In this case the communication at each step does not form a matching. Instead, we can view it as a directed graph, with each node having indegree and outdegree at most one. We were able to develop a 6 approximation for the data migration problem under this model [15]. In addition in most of the data migration papers it is assumed that a connection can be established between any pair of storage devices and that data can be exchanged at the same rate between any pair of storage devices. For a heterogeneous distributed system this assumption is not realistic. In special cases such as for broadcasting and multicasting we have been able to develop approximation algorithms

that work well [14, 16] in such models. In fact the algorithms that we analyze are very simple methods, and we show that they produce solutions that are close to optimal (the proofs are somewhat involved). We do not have any bounds for general data migration problems on such models.

5. The final challenge is in making this work in real-life large scale systems. As part of this evaluation effort, we would also like to explore techniques for evaluating our algorithms' and system's ability to self-adjust to system faults.

6. Finally, we completely ignored the issue of fairness. For example, currently we do not have any client importance levels or QoS specifications for the clients. In multimedia applications, the bandwidth rate can be adjusted with certain degradation in the quality of the video. The next step would be to incorporate that into the data layout algorithms. So for example, more than L data streams can be handle by a disk by lowering the quality of the transmission.

References

[1] Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin. *Network flows: theory, algorithms, and applications*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1993.

[2] E. Anderson, J. Hall, J. D. Hartline, M. Hobbs, A.Karlin, J. Saia, R. Swaminathan, and J. Wilkes. An experimental study of data migration algorithms. In *WAE '01: Proceedings of the 5th International Workshop on Algorithm Engineering*, pages 145–158, London, UK, 2001. Springer-Verlag.

[3] V. Arya, N. Garg, R. Khandekar, V. Pandit, A. Meyerson, and K. Munagala. Local search heuristics for k-median and facility location problems. *SIAM J. on Computing*, 33(3):544–562, 2004.

[4] Ivan D. Baev and Rajmohan Rajaraman. Approximation algorithms for data placement in arbitrary networks. In *SODA '01: Proceedings of the twelfth annual ACM-SIAM symposium on Discrete algorithms*, pages 661–670, Philadelphia, PA, USA, 2001. Society for Industrial and Applied Mathematics.

[5] L. Golubchik, S. Khanna, S. Khuller, R. Thurimella, and A. Zhu. Approximation algorithms for data placement on parallel disks. In *SODA '00: Proceedings of the 11th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 223–232, Philadelphia, PA, USA, 2000. Society for Industrial and Applied Mathematics.

[6] L. Golubchik, S. Khuller, Y. Kim, S. Shargorodskaya, and Y-C. Wan. Data migration on parallel disks. In *Proc. of European Symp. on Algorithms (2004). LNCS 3221*, pages 689–701. Springer, 2004.

[7] S. Guha and K. Munagala. Improved algorithms for the data placement problem. In *Proceedings of the 13th ACM-SIAM Symposium on Discrete Algorithms*, 2002.

[8] J. Hall, J. Hartline, A. Karlin, J. Saia, and J. Wilkes. On algorithms for efficient data migration. In *SODA '01: Proceedings of the twelfth annual ACM-SIAM symposium on Discrete algorithms*, pages 620–629, Philadelphia, PA, USA, 2001. Society for Industrial and Applied Mathematics.

[9] J. Jung, B. Krishnamurthy, and M. Rabinovich. Flash crowds and denial of service attacks: Characterization and implications for cdns and web sites, 2002.

[10] S. Kashyap and S. Khuller. Algorithms for non-uniform size data placement on parallel disks. In *Proc. of Foundations of Software technology and Theoretical Computer Science, LNCS 2914*, pages 265–276. Springer, 2003.

[11] S. Kashyap, S. Khuller, Y-C. Wan, and L. Golubchik. Fast reconfiguration of data placement in parallel disks. In *Proc. of ALENEX Workshop*. SIAM, 2006.

[12] S. Khuller, Y. Kim, and Y-C. Wan. Algorithms for data migration with cloning. In *PODS '03: Proceedings of the 22ndACM symposium on Principles of database systems*, pages 27–36, New York, NY, USA, 2003. ACM Press.

[13] S. Khuller, Y. Kim, and Y-C. Wan. On generalized broadcasting and gossiping. In *Proc. of the European Symp. on Algorithms*, pages 373–384. Spring Verlag, 2003.

[14] S. Khuller and Y.A. Kim. Broadcasting on heterogenous networks. In *Proceedings of the ACM-SIAM symposium on Discrete Algorithms*, pages 1004–1013. ACM, 2004.

[15] S. Khuller, Y.A. Kim, and A. Malekian. Improved algorithms for data migration. In *APPROX: Workshop on Approximation Algorithms*. LNCS, 2006.

[16] S. Khuller, Y.A. Kim, and Y-C. Wan. On broadcasting on networks of workstations. In *SPAA: Proceedings of the ACM symposium on Parallel Algorithms and Architectures*. ACM, 2005.

[17] M. R. Korupolu, C. G. Plaxton, and R. Rajaraman. Placement algorithms for hierarchical cooperative caching. In *SODA '99: Proceedings of the 10th annual ACM-SIAM symposium on Discrete algorithms*, pages 586–595, Philadelphia, PA, USA, 1999. Society for Industrial and Applied Mathematics.

[18] H. Shachnai and T. Tamir. On two class-constrained versions of the multiple knapsack problem. *Algorithmica*, 29(3):442–467, 2001.