Approximation Algorithms for Data Placement on Parallel Disks

LEANA GOLUBCHIK
Department of Computer Science
University of Southern California, Los Angeles
and
SANJEEV KHANNA
Department of Computer and Information Science
University of Pennsylvania, Philadelphia
and
SAMIR KHULLER
Department of Computer Science and Institute for Advanced Computer Studies
University of Maryland, College Park
and
RAMAKRISHNA THURIMELLA
Department of Computer Science
The University of Denver, Denver
and
AN ZHU
Google Inc, Mountain View.

A preliminary version of this paper was presented at the ACM-SIAM Symposium on Discrete Algorithms (2000).
Contact Information:
L. Golubchik, E-mail : leana@cs.usc.edu. This work was done while this author was at the University of Maryland. Research supported by NSF CAREER Award CCR-9896232.
S. Khanna, Email: sanjeev@cis.upenn.edu. Supported in part by an Alfred P. Sloan Research Fellowship and an NSF Career Award CCR-0093117.
S. Khuller, E-mail : samir@cs.umd.edu. Research supported by NSF Award CCR-9820965 and by NSF CCR-0113192.
R. Thurimella, E-mail : ramki@cs.du.edu. This work was done while this author was visiting the University of Maryland.
A. Zhu, Email : anzhu@cs.stanford.edu. This work was done while this author was at the University of Maryland and supported by NSF Award CCR-9820965.
Permission to make digital/hard copy of all or part of this material without fee for personal or classroom use provided that the copies are not made or distributed for profit or commercial advantage, the ACM copyright/server notice, the title of the publication, and its date appear, and notice is given that copying is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists requires prior specific permission and/or a fee.
© 20YY ACM 0000-0000/20YY/0000-0001 $5.00

We study an optimization problem that arises in the context of data placement in a multimedia storage system. We are given a collection of M multimedia objects (data objects) that need to be assigned to a storage system consisting of N disks $d_1, d_2 ..., d_N$. We are also given sets $U_1, U_2, ..., U_M$ such that U_i is the set of clients seeking the ith data object. Each disk d_j is characterized by two parameters, namely, its *storage capacity* C_j which indicates the maximum number of data objects that may be assigned to it, and a *load capacity* L_j which indicates the maximum number of clients that it can serve. The goal is to find a placement of data objects to disks and an assignment of clients to disks so as to maximize the total number of clients served, subject to the capacity constraints of the storage system.

We study this data placement problem for two natural classes of storage systems, namely, *homogeneous* and *uniform ratio*. We show that an algorithm developed by [Shachnai and Tamir, *Algorithmica*, 29(3):442–467] for data placement, achieves the *best possible* absolute bound regarding the number of clients that can always be satisfied. We also show how to implement the algorithm so that it has a running time of $O((N + M) \log(N + M))$. In addition, we design a polynomial time approximation scheme, solving an open problem posed in the same paper.

Categories and Subject Descriptors: F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems

General Terms: Algorithms, Theory

Additional Key Words and Phrases: Data Placement, Storage Systems, Approximation algorithms

1. INTRODUCTION

We study a *data placement problem* that arises in the context of multimedia storage systems. In this problem, we are given a collection of M multimedia objects (data objects) that need to be assigned to a storage system consisting of N disks $d_1, d_2 ..., d_N$. We are also given pairwise disjoint sets $U_1, U_2, ..., U_M$ such that U_i is the set of clients seeking the ith data object. Each disk d_j is characterized by two parameters, namely, its *storage capacity* C_j which indicates the maximum number of data objects that may be assigned to it, and its *load capacity* L_j which indicates the maximum number of clients that it can serve. The goal is to find a placement of data objects to disks and an assignment of clients to disks so as to maximize the total number of clients served, subject to the capacity constraints of the storage system.

The data placement problem described above arises naturally in the context of storage systems for multimedia objects where one seeks to find a placement of the data objects such as movies on a system of disks. We study our data placement problem for the following two natural types of storage systems. (In Subsection 1.3 we will indicate how such systems arise by "grouping" together heterogeneous disks.)

Homogeneous Storage Systems: In a homogeneous storage system, all disks are identical. We denote by k and L the storage capacity and the load capacity, respectively, of each disk and refer to this variant as k-HDP (homogeneous data placement).

Uniform Ratio Storage Systems: In a uniform ratio storage system, the ratio L_j/C_j of the load to the storage capacity is identical for each disk. We denote by $C_{\min}$ and $C_{\max}$ the minimum and the maximum storage capacity of any disk in such a system and refer to this variant as URDP (uniform ratio data placement).

In the remainder of this paper, we assume without loss of generality that (i) the

total number of clients does not exceed the total load capacity, i.e., $\sum_{i=1}^{M} |U_i| \leq \sum_{j=1}^{N} L_j$, and (ii) the total number of data objects does not exceed the total storage capacity, i.e., $M \leq \sum_{j=1}^{N} C_j$.

1.1 Related Work

The data placement problem described above bears some resemblance to the classical multi-dimensional knapsack problem [Frieze and Clarke 1984; Raghavan 1988; Chekuri and Khanna 1999]. We can view each disk as a knapsack with a load as well as a storage dimension, and each client as a unit size item with a color associated with it. Items have to be packed in knapsacks in such a way that the number of items in a knapsack does not exceed L_j, its load capacity. Knapsacks will have an additional color constraint, namely that the total number of distinct colors of items assigned to it does not exceed C_j its storage capacity. However, in our problem, the storage dimension of a disk behaves in a *non-aggregating* manner in that assigning additional items of an already present color does not increase the load along the storage dimension. It is this distinguishing aspect of our problem that makes it difficult to apply known techniques for multi-dimensional packing problems.

Shachnai and Tamir [Shachnai and Tamir 2000a] studied the above data placement problem; they refer to it as the *class constrained multiple knapsack* problem. The authors gave an elegant algorithm, called the *Sliding-Window* algorithm, and showed that this algorithm packs all items whenever $\sum_{j=1}^{N} C_j \geq M + N - 1$ for URDP. An easy corollary of this result is that one can always pack a $(1 - \frac{1}{1+C_{\min}})$-fraction of all items for URDP. The authors showed that the problem is NP-hard when each disk has an arbitrary load capacity, and unit storage. Subsequent to our work, Shachnai and Tamir [Shachnai and Tamir 2000b] studied a variation of the data placement problem above where in addition to having a color, each item u has a size $s(u)$ and a profit $p(u)$ associated with it. For the special case when $s(u) = p(u)$ for each item, and the *total* number of different colors (M) is *constant*, the authors give a dual approximation scheme whereby for any $\epsilon > 0$, they give a polynomial time algorithm to obtain a $(1 - \epsilon/4)$-approximate solution provided the load capacity is allowed to be exceeded by a factor of $(1 + \epsilon)$.

After the publication of a preliminary version of this paper, some of the results presented were extended to the case when the data objects do not all have unit size [Kashyap and Khuller 2003; Shachnai and Tamir 2003].

1.2 Our Results

Our first main result is a tight upper and lower bound on the number of items that can *always* be packed for any input instance to homogeneous as well as uniform ratio storage systems, regardless of the distribution of requests for data objects. It is worth noting that in the case of arbitrary storage systems no such absolute bounds are possible.

THEOREM 1.1. (Section 3) *It is always possible to pack a $(1 - \frac{1}{(1+\sqrt{k})^2})$-fraction of items for any instance of k-HDP, or more generally, a $(1 - \frac{1}{(1+\sqrt{C_{\min}})^2})$-fraction of items can always be packed for any instance of URDP. Moreover, there exists a family of instances for which it is infeasible to pack any larger fraction of items.*

The upper bounds above are achieved constructively, by a tight analysis of the sliding window algorithm of [Shachnai and Tamir 2000a]. A side-result of our proof technique here is a simple alternate proof of the result that all items can be packed whenever $\sum_{j=1}^{N} C_j \geq M + N - 1$.

Our second main result is a polynomial time approximation scheme (PTAS) for the data placement problem in uniform ratio storage systems, answering an open question of [Shachnai and Tamir 2000a].

THEOREM 1.2. (Section 4) *For any fixed* $\epsilon' > 0$, *one can obtain in polynomial time a* $(1-\epsilon')$*-approximate solution to the data placement problem for any instance of* URDP.

We also strengthen the NP-hardness results of [Shachnai and Tamir 2000a] by showing that the data placement problem is NP-hard even for very special cases of homogeneous storage systems.

THEOREM 1.3. (Appendix A) *The* k-HDP *problem is NP-complete for homogeneous disks with storage capacity* $k = 2$ *and strongly NP-hard for* $k = 3$.

Both reductions above are from NP-hard partitioning problems and illustrate how item colors can effectively encode large non-uniform sizes arising in the instances of these partitioning problems, even though each item in our problem is unit size itself. We also note here that the case $k = 1$ is easily solvable in polynomial time using a simple greedy algorithm in which we first pack items of the most popular color.

Finally, we also study the problem from an empirical perspective. We study the homogeneous case on instances generated by a Zipf distribution [Knuth 1973] (this corresponds to *measurements* performed in [Chervenak 1994] for a movies-on-demand application) and compare the actual performance of the Sliding-Window algorithm with the bounds obtained above as well as the bounds in [Shachnai and Tamir 2000a]. The results of this study are presented in Appendix B (see Figs. 2, 3, 4). We also show how to implement the Sliding-Window algorithm so that it runs in $O((N + M)\log(N + M))$ steps, improving on the $O(NM)$ implementation described in [Shachnai and Tamir 2000a].

We next describe in some detail the motivating application for our data placement problem.

1.3 Motivational Application

Recent advances in high speed networking and compression technologies have made multimedia services feasible. Take for instance, the video-on-demand(VOD) servers. The enormous storage and bandwidth requirements of multimedia data necessitates that such systems have very large disk farms. One viable architecture is a parallel (or distributed) system with multiple processing nodes in which each node has its own collection of disks and these nodes are interconnected, e.g., via a high-speed network.

We note that disks are a particularly interesting resource. Firstly, disks can be viewed as "multidimensional" resources, the dimensions being storage capacity and load capacity, where depending on the application one or the other resource can be the bottleneck. Secondly, all disk resources are not equivalent since a disk's utility

is determined by the data stored on it. It is this "partitioning" of resources (based on data placement) that contributes to some of the difficulties in designing cost-effective parallel multimedia systems, and I/O systems in general. In a large parallel VOD system improper data distribution can lead to a situation where requests for (popular) videos cannot be serviced even when the overall load capacity of the system is not exhausted because these videos reside on highly loaded nodes, i.e., the available load capacity and the necessary data are not on the same node.

One approach to addressing the load imbalance problem is to partition each video across all the nodes in the system and thus avoid the problem of "splitting resources", e.g., as in the staggered striping technique [Berson et al. 1994]. However, this approach suffers from a number of implementation-related shortcomings that are detailed in [Chou et al. 2002]. An alternate system is described in [Wolf et al. 1995] where the nodes are connected in a shared-nothing manner [Stonebraker 1986]. Each node j has a finite storage capacity, C_j (*in units of continuous media (CM) objects*), as well as a finite load capacity, L_j (*in units of CM access streams*). These nodes are constructed by putting together several disks. In fact, in the paper we will mostly view nodes as "logical disks". For instance, consider a server that supports delivery of MPEG-2 video streams where each stream has a bandwidth requirement of 4 Mbits/s and each corresponding video file is 100 mins long. If each node in such a server has 20 MBytes/s of load capacity and 36 GB of storage capacity, then each such node can support $L_j = 40$ simultaneous MPEG-2 video streams and store $C_j = 12$ MPEG-2 videos. In general, different nodes in the system may differ in their storage and/or load capacities.

In our system each CM object resides on one or more nodes of the system. The objects may be striped on the *intra-node* basis but *not* on the *inter-node* basis. Objects that require more than a single node's load capacity (to support the corresponding requests) are *replicated* on multiple nodes. The number of replicas needed to support requests for a continuous object is a function of the demand. This should result in a scalable system which can grow on a node by node basis.

The difficulty here is in deciding on: (1) how many copies of each video to keep, which can be determined by the demand for that video, as in [Wolf et al. 1995], and (2) how to place the videos on the nodes so as to satisfy the total anticipated demand for each video within the constraints of the given storage system architecture. It is these issues that give rise to our data placement problem.

1.4 Organization

We start with an overview of the Sliding-Window algorithm in Section 2. In Section 3, we present a tight analysis of the Sliding-Window algorithm to derive the upper bounds of Theorem 1.1. We also present here a family of instances that give the matching lower bound. Finally, in Section 4 we present our approximation schemes for homogeneous as well as uniform ratio storage systems and thus establish Theorem 1.2. We defer to the Appendix a proof of Theorem 1.3 as well as the details of implementation issues and our empirical results.

2. SLIDING-WINDOW ALGORITHM

For completeness we describe the algorithm [Shachnai and Tamir 2000a] that applies to both the homogenous case and the uniform ratio case. Recall that data objects

can be viewed as colors. Each client is an item with a color. So essentially, we have groups of items with colors. Let $R[i]$ be number of items of color i. We order the colors so that $R[1] \leq R[2] \leq \ldots \leq R[M]$.

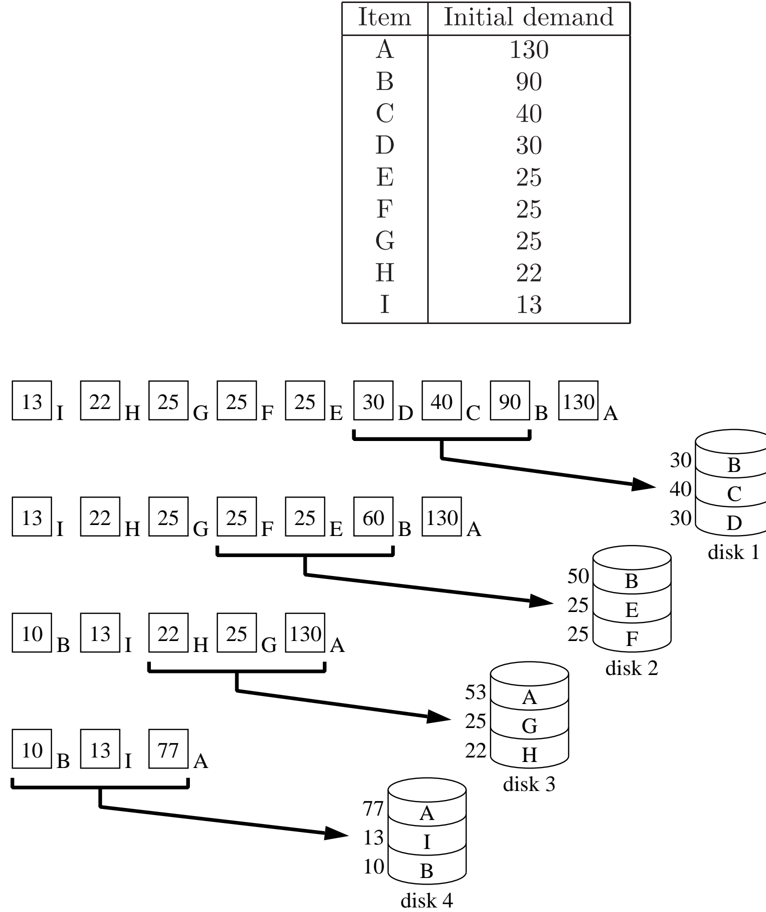
We keep all the colors in a sorted list in non-decreasing order of the number of items of that color, denoted by R. The list, $R[1], \ldots, R[m]$, $1 \leq m \leq M$, is updated during the algorithm. At step j, we assign items to disk d_j. For the sake of notation simplification, $R[i]$ always refers to the number of *currently* unassigned items of a particular color (i.e., we do not explicitly indicate the current step j of the algorithm in this notation). We order the knapsacks in non-decreasing capacity order, i.e., $C_1 \leq C_2 \leq \cdots \leq C_N$. We assign items and remove from R the colors that are packed completely, and we move (at most) one partially packed color to its updated place according to the remaining number of unpacked items of that color.

The assignment of colors to disk d_j follows the general rule that we want to select the first consecutive sequence of C_j or less colors, $R[u], \ldots, R[v]$, whose total number of items either equals to or exceeds the load capacity L_j. We then assign items of colors $R[u], \ldots, R[v]$ to d_j. In order to not exceed the load capacity, we will split the items of the last color $R[v]$. It could happen that no such sequence of colors is available, i.e., all colors have relatively few items. In this case, we greedily select the colors with the largest number of items to fill the current disk. In particular, the selection procedure is as follows: we first examine $R[1]$, which is the color with the smallest number of items. If these items exceed the load capacity, we will assign $R[1]$ to the first disk and re-locate the remaining piece of $R[1]$ (which for R[1] will always be the beginning of the list). If not, we then examine the total number of items of $R[1]$ and $R[2]$, and so on until either we find a sequence of at most C_j colors with a sufficiently large number of items ($\geq L_j$), or the first C_j colors have a total number of items $< L_j$. In the latter case, we go on to examine the next C_j colors $R[2], \ldots, R[C_j + 1]$ and so on, until either we find C_j colors with a total number of items at least L_j or we are at the end of the list, in which case we simply select the last sequence of C_j colors which has the greatest total number of items.

We show how to implement the algorithm to run in $O((N+M)\log(N+M))$ steps, where N is the number of disks and M is the number of colors. Note that this is a significant improvement on the running time in [Shachnai and Tamir 2000a]. Appendix B provides the pseudo code for the Sliding-Window algorithm and necessary data structures realizing the faster implementation.

In this toy example, we consider a storage system that consists of four identical disks. Each disk has storage capacity of three units and load capacity of 100 units. There are nine colors that need to be stored in the system. The number of items for each color is as shown.

We describe how the first disk is packed. Since the disk has storage capacity 3, we initialize a window of size 3 at the beginning of the list. The total number of items corresponding to this set of colors is only 60, which is lower than the load capacity of 100. We then slide the window, and the first subset of 3 colors that have at least 100 items is the set $\{D, C, B\}$. However we only pack 30 items of color B, and the remaining 60 items are re-inserted into the list at the correct position. In this example, all the items get packed and nothing is left out (see Fig. 2).



Item	Initial demand
A	130
B	90
C	40
D	30
E	25
F	25
G	25
H	22
I	13

Fig. 1. Storage capacity k=3, Bandwidth $L = 100$. In addition to producing the layout the sliding window algorithm finds a mapping of items to disks, which is optimal for the layout computed.

3. ANALYSIS

We first show that the Sliding-Window algorithm guarantees to pack $(1 - \frac{1}{(1+\sqrt{k})^2})$ fraction of items in the homogenous case. We assume each disk has load capacity L and storage capacity k.

Note that if there are some unpacked items, then every disk is filled to the maximum either on the number of items it can handle or on the number of colors that can be stored. We will call the former as *load saturated* and the latter (the rest) as *storage saturated*. (Therefore, if a disk is storage saturated, then it still has some unfilled load capacity.) Denote the number of load-saturated disks and the number of storage-saturated disks by N_L and N_S, respectively. It is easy to see that $d_1, \ldots, d_{N_L}$ are load-saturated disks, and the rest are storage-saturated disks. Let m_j denote the number of colors assigned to disk d_j. Obviously for storage-saturated disks, $m_j = k$. Let $c < 1$ be the smallest fraction of load to which a

storage-saturated disk is filled. Note that this disk must store a color with a number of items of that color being at most $c \times L/k$. (Minimum is at most the average.) Now every color on the unassigned list has no more than $c \times L/k$ remaining items of that color. Otherwise, the Sliding-Window algorithm would have put this color on the lightest-loaded disk and increase greedily the total number of packed items. We define the notion of "splitting a color." This refers to the situation where only some of the items of a particular color are packed into the current disk; while the remaining items are left behind to be considered further by the algorithm. Hence, the particular color is split and might be packed into multiple disks.

LEMMA 3.1. *Using the Sliding-Window algorithm, the number of unpacked items is at most* $\frac{c \times L \times N_L}{k}$.

PROOF. The main thing we need to prove is that there are at most N_L colors left after we run the Sliding-Window algorithm. For each left over color we know that the number of items is at most $\frac{c \times L}{k}$, so the total number of unpacked items is at most $\frac{c \times L \times N_L}{k}$.

We examine the number of colors stored in the load-saturated disks. If there is a load-saturated disk d_j with $m_j < k$ colors, then there are no colors left when the algorithm terminates, i.e., all items are packed. This can be explained as follows. The reason that less than k colors are packed into d_j is due to the fact that at step j $\sum_{i=1}^{m_j} R[i] \geq L$. Since we sort the colors in non-decreasing order of number of items, at this point any consecutive sequence of $k-1$ colors in the list has the total size $\geq L$. Since at step j, we "split" at most one color, which is always added to the beginning of R. At any step $t \geq j$ we have a guarantee that, for the new list R, $\sum_{i=1}^{k} R[i] \geq L$, unless we have less than k colors. This implies that we fill the disks to their load capacity until we run out of colors. Hence we can pack all items.

We can now assume that all the load-saturated disks have k colors. The storage-saturated disks have k colors as well. We start with $M \leq N \times k$ colors. During the process, we can split at most N_L colors, i.e., we can generate at most N_L new "instances" of originally existing colors. This is because only filling disks that are load-saturated can result in generating new "instances" of colors. So the number of new "instances" of colors generated is upper bounded by the number of load-saturated disks. Thus the number of colors left is $\leq M + N_L - N \times k \leq N_L$. □

LEMMA 3.2. *Using the Sliding-Window algorithm, the number of unpacked items is at most* $(1-c) \times L \times N_S$.

PROOF. At least $L \times N_L + c \times L \times N_S$ items are packed. Subtracting this quantity from an upper bound on the total number of items $N \times L$ gives $N_S \times L + N_L \times L - L \times N_L - c \times L \times N_S$. This proves the claim. □

THEOREM 3.3. *The Sliding-Window algorithm guarantees to pack* $(1 - \frac{1}{(1+\sqrt{k})^2})$ *fraction of items in the homogenous case.*

PROOF. The above two lemmas give us two upper bounds on the number of unpacked items. The number of unpacked items is at most $\min(\frac{c \times L \times N_L}{k}, (1-c) \times$

$L \times N_S$). The number of packed items is at least $L \times N_L + c \times L \times N_S$. We show that the ratio of unpacked(U) to packed(S) items is at most

$$\frac{U}{S} \leq \frac{\min(\frac{c \times L \times N_L}{k}, (1-c) \times L \times N_S)}{L \times N_L + c \times L \times N_S}.$$

Let $y = \frac{N_L}{N}$ and thus $\frac{N_S}{N} = 1 - y$. Simplifying the upper bound for the number of unpacked to packed items, we obtain

$$\frac{\min(\frac{cy}{k}, (1-c)(1-y))}{y + c(1-y)}.$$

This is the same as

$$\min(\frac{\frac{cy}{k}}{y + c(1-y)}, \frac{(1-c)(1-y)}{y + c(1-y)}).$$

We can simplify the two functions to the following

$$\min(\frac{1}{k \times (\frac{1}{c} + \frac{1-y}{y})}, \frac{(1-c)(1-y)}{1 - (1-c)(1-y)}).$$

The first term is strictly increasing as c or y increases, while the second term is strictly decreasing as c or y increases. So in order to maximize the expression, the two terms should be equal, which means

$$\frac{cy}{k} = (1-c)(1-y).$$

This gives

$$y = \frac{1-c}{1 - c + \frac{c}{k}}.$$

Substituting for y gives us that the upper bound for $\frac{U}{S}$ to be at most

$$\frac{c - c^2}{k - kc + c^2}.$$

This achieves its maxima when $c = (1 - \frac{1}{1+\sqrt{k}})$.
The fraction of all items that are packed is

$$\frac{S}{U+S} = \frac{1}{1 + \frac{U}{S}}.$$

Replacing the bound that we derived for c we get that

$$\frac{U}{S} \leq \frac{1}{k + 2\sqrt{k}}.$$

This yields

$$\frac{S}{U+S} \geq 1 - \frac{1}{(1+\sqrt{k})^2}.$$

which proves the claim. □

Corollary 3.4. *For "uniform ratio" disks, if $\sum_{j=1}^{N} C_j \geq M + N - 1$, then all colors can be packed using the Sliding-Window algorithm.*

Proof. This is an alternate proof for the claim in [Shachnai and Tamir 2000a]. Our analysis of the algorithm makes the proof simpler.

Let $r = \frac{L_j}{C_j}$, denote the uniform ratio. Since the ratios' $\frac{L_j}{C_j}$ are uniform, once any disk becomes storage-saturated, the rest of the disks will be storage-saturated as well. The main claim we need to prove is that after we fill disk d_{N-1}, we have at most C_N colors left. We will prove this shortly. If d_{N-1} is storage-saturated, then we can safely assign the remaining C_N colors to d_N. If d_{N-1} is load-saturated, then all previous disks are load-saturated. Since the total number of items does not exceed the total load capacity, we will not exceed the load capacity.

We argue that if there is a load-saturated disk d_j with $m_j < C_j$, then all the items will be packed. At this stage, $R[\ell] \geq \frac{L_j}{m_j} \geq \frac{C_j}{C_j - 1} \times r \geq \frac{C_t}{C_t - 1} \times r$ for all $\ell > m_j$ and all $t \geq j$. Recall that disks are sorted in non-decreasing order of C_i. Thus we have the following result: at any step $t > j$, $\sum_{i=1}^{C_t} R[i] \geq \sum_{i=2}^{C_t} R[i] \geq C_t \times r = L_t$, and so all items are packed without sliding the window. Since all load-saturated disks have C_j colors, after we fill disk d_{N-1}, we have generated at most $N-1$ new "instances" of colors. The total number of colors left is $\leq M + N - 1 - \sum_{i=1}^{N-1} C_i \leq C_N$. □

We now extend the above proof to the uniform-ratio case. The motto in the homogenous case is that the higher the disk's storage capacity the better the performance of the Sliding-Window algorithm. So in a uniform-ratio system one should expect the algorithm to do at least as well as in the homogenous case where all disks assume the smallest disk size in the uniform-ratio system. The following theorem formally proves this intuition.

Theorem 3.5. *Let C_{min} denote the minimum capacity of a disk in the uniform ratio system. The Sliding-Window algorithm guarantees to pack $(1 - \frac{1}{(1+\sqrt{C_{\min}})^2})$ fraction of the total items.*

Proof. Let $r = \frac{L_j}{C_j}$ for $j = 1 \ldots N$ denote the uniform ratio. From the proof of Corollary 3.4 above, if there is a load-saturated disk d_j with $m_j < C_j$, then all items will be packed. Thus we will focus on the case where for all j we have $m_j = C_j$. Let M_L denote the total number of colors (we count the same colors in different disks as different multiple colors) in the load-saturated disks $1, \ldots, N_L$, so $M_L = \sum_{j=1}^{N_L} m_j = \sum_{j=1}^{N_L} C_j$. Let M_S denote the total number of colors in the storage-saturated disks $N_L + 1, \ldots, N$, so $M_S = \sum_{j=N_L+1}^{N} m_j = \sum_{j=N_L+1}^{N} C_j$. Again let c be the smallest fraction of load to which a storage-saturated disk is filled. Thus we have the following similar results:

(1) For each left over color we know that the number of items is at most $c \times r$.

(2) There are at most N_L colors left unassigned. We have $N_L \leq \frac{M_L}{m_{min}} = \frac{M_L}{C_{\min}}$.

(3) The number of unpacked items is then at most $c \times r \times \frac{M_L}{C_{\min}}$.

(4) At least $r \times M_L + c \times r \times M_S$ items are packed.

(5) The number of unpacked items is at most $(1 - c) \times r \times M_S$.

Hence the ratio of the unpacked(U) to packed(S) items is at most

$$\frac{U}{S} \leq \frac{\min(\frac{c \times r \times M_L}{C_{\min}}, (1-c) \times r \times M_S)}{r \times M_L + c \times r \times M_S}.$$

Let $y = \frac{M_L}{M_L + M_S}$ and thus $\frac{M_S}{M_L+M_S} = 1 - y$. Simplifying the upper bound for the expression, we obtain

$$\frac{U}{S} \leq \frac{\min(\frac{cy}{C_{\min}}, (1-c)(1-y))}{y + c(1-y)}.$$

Note that this is the same expression as in Theorem 3.3 for the homogenous system. Optimizing this expression gives the same bound as in Theorem 3.3 with k replaced by $C_{\min}$. This proves the claim. □

3.1 Tight Examples

We now give an example to show that the bound of $(1 - \frac{1}{(1+\sqrt{k})^2})$ is tight. In other words, there are instances for which no solution will pack more than $(1 - \frac{1}{(1+\sqrt{k})^2})$ fraction of items.

The trivial case is when k, the storage capacity of a disk, is 1. Consider $N = 2$ disks, with $L = 2$, and two colors having 1 and 3 items respectively. A simple check shows we can pack at most 3 items. Now consider the case where k is a perfect square and $k \geq 2$. Let N, the number of disks, be $1+\sqrt{k}$, and let $L = k+\sqrt{k}$. There are $\sqrt{k}$ colors with a large number of items each, $U_1, \ldots, U_{\sqrt{k}}$ with $|U_i| = 2+\sqrt{k}$ for $1 \leq i \leq \sqrt{k}$; we will refer to these as "large colors". And, there are $(k-1)(1+\sqrt{k})+1$ colors with a small number of items each, $U_{\sqrt{k}+1}, \ldots, U_{k(1+\sqrt{k})}$ with $|U_i| = 1$ for $\sqrt{k}+1 \leq i \leq k(1+\sqrt{k})$; we will refer to these as "small colors".

We will show that there are always at least $\sqrt{k}$ items that do not get packed. In this case, the fraction of items that are not packed is at least $\frac{\sqrt{k}}{(1+\sqrt{k})(k+\sqrt{k})}$ which is exactly $\frac{1}{(1+\sqrt{k})^2}$. This proves the claim.

We first consider the $\sqrt{k}$ large colors. An unsplit set U_i has all its items packed in a single disk. A split set U_i has its items packed in several disks. For a disk that contains at least one large unsplit color, the available load capacity left is at most $k-2$. (Note that after packing one large unsplit color, the available load capacity is smaller than the storage capacity.) For any of the remaining large color on this disk with $l \geq 2$ items, we can exchange the color with any l (distinct) small colors in any other disk, while still packing the same number of items. The disk now have one large unsplit color, and at most $k-2$ small colors. We perform such exchange for each disk containing an unsplit large color. The remaining disks have only large split colors. In fact, assume that there are exactly p $(0 \leq p \leq \sqrt{k})$ large colors that do not get split $U_1, \ldots, U_p$, with disk d_i containing U_i.

Now consider the remaining $N-p$ disks; we are left with at least $k \times N - p(k-1) = k \times (N-p) + p$ colors, but we only have $k \times (N-p)$ storage capacity left. Since the remaining $\sqrt{k} - p$ large colors are all split, this generates an additional $\sqrt{k} - p$

"instances" of colors. Thus we have at least $k \times (N - p) + p + \sqrt{k} - p$ colors. This will create an excess of $\sqrt{k}$ items that cannot be packed.

We now extend the above example to the case where k is not necessarily a perfect square. We show that the targeted bound $(1 - \frac{1}{(1+\sqrt{k})^2})$ can be approximated arbitrarily close by a rational number. Recall the proof of Theorem 3.3, where the setting of the parameters $c = 1 - \frac{1}{1+\sqrt{k}}$ and $y = \frac{1-c}{1-c+\frac{c}{k}}$ achieves the desired bound. We will approximate c with a rational number. Let $c \approx \frac{p}{q}$, with integers p and q co-prime. The choice of p and q is arbitrary, and we may pick such p and q so that $\frac{p}{q}$ closely approximate $(1 - \frac{1}{1+\sqrt{k}})$. Substitute c in the expression of y, we know $y = \frac{1-c}{1-c+\frac{c}{k}} = \frac{kq-kp}{kq-kp+p}$. From the value of y, we let $N = kq-kp+p$, so $N_L = kq-kp$ and $N_S = p$. Each small color has exactly $\frac{Lp}{kq}$ items. To simplify matters, we assume $L = kq$, so $\frac{Lp}{kq} = p$. Each large color has exactly $kq - kp + 2p$ items. So it is still true that the total load of $k - 2$ small colors and one large color adds up to L. We have $kq - kp$ large colors and $(kq - kp)(k - 1) + pk$ small colors. With a simple calculation, we can verify that the total load is exactly $kq \times (kq - kp + p) = LN$, and total number of colors is exactly $k \times (kq - kp + p) = kN$. We aim to show that at most $kq \times (kq - kp) + kp \times p$ items can be packed by any algorithm. In this case, the ratio of unpacked (U) to packed (S) items is at least

$$\frac{(kq - kp)p}{kq(kq - kp) + kp(p)} = \frac{(1 - \frac{p}{q})\frac{p}{q}}{k - k \cdot \frac{p}{q} + (\frac{p}{q})^2}.$$

We first prove a couple of useful lemmas to shape the structure of the optimal packing.

LEMMA 3.6. *If all small colors are partially assigned in the packing $\mathcal{P}$, then $\mathcal{P}$ packs no more than $kq \times (kq - kp) + kp \times p$ items.*

PROOF. Since originally there are $(kq-kp)(k-1)+pk$ small colors, by the pigeon hole principle, at least p disks contain k small colors each. Thus the total items on each of these disks is no more than kp, and the total items packed overall is no more than $(kq - kp)kq + kp \times p$. □

From now on we will only concern ourselves with instances where every optimal packing has at least one unassigned small color.

LEMMA 3.7. *There exists an optimal packing of the above instance, such that no small colors are split.*

PROOF. Proof by contradiction. Suppose every optimal packing of the above instance has to split some small colors. Choose the optimal packing that splits the least number of small colors. Pick any split small color U_s in disk d_j, for some small color U_s. We modify the packing by assigning all items of U_s to d_j. If the load capacity is not violated by such modification, then we are done. We have created a packing of the same value with one less split small color. If the modification leads to a total load bigger than L, then we argue that there exists a large color U_l on d_j with more than p items. Otherwise, all colors on d_j have no more than p items, a maximum of k such colors will never exceed the load capacity. We simply truncate the number of items of U_l assigned to d_j, so that the resulting load is exactly L.

The truncated items can be allocated to where the remaining items of U_s were assigned originally. Again, we have created a packing of the same value with one less split small color; this proves the claim. □

From now on we only concern ourselves with optimal packings, which in addition to having at least one unassigned small color, do not split any small colors.

LEMMA 3.8. *There exists an optimal packing of the above instance, such that no large color is partially assigned to a disk with number of items between* $(0, p)$.

PROOF. Proof by contradiction. Consider an optimal packing $\mathcal{P}$ that packs the most number of small colors. Assume in $\mathcal{P}$ there exist a large color U_l and disk d_j such that there are between $(0, p)$ number of items of color U_l on d_j. We have assumed there is at least one unassigned small color U_s. We assign U_s completely to d_j, and remove the items of color U_l. If the modification leads to a total load bigger than L, then we argue that there exists a large color $U_{l'} \neq U_l$ on d_j with more than p items as in Lemma 3.7. We simply truncate the number of items of $U_{l'}$ assigned to d_j, so that the resulting load is exactly L. Notice that in the created packing, no small color is split. Now either we have created a packing with one less unassigned small color, contradicting the choice of $\mathcal{P}$; or the resulting packing has no unassigned small color, contradicting the claim that every optimal packing has at least one unassigned small color. □

From now on we only concern ourselves with optimal packings, which in addition to the properties mentioned earlier, have no large color partially assigned to a disk with number of items between $(0, p)$.

LEMMA 3.9. *There exists an optimal packing of the above instance, such that no large color is partially assigned to a disk with a number of items between* $(kq - kp + p, kq - kp + 2p)$.

PROOF. Proof by contradiction. Consider the optimal packing that has the least number of such color disk pairs. Let U_l be a large color partially assigned to a disk d_j with number of items between $(kq - kp + p, kq - kp + 2p)$. Since there are less than p items of color U_l left, by the previous lemma, all these remaining items are unassigned. Besides U_l, there are $k - 1$ remaining storage spaces, and between $((k-2)p, (k-1)p)$ remaining load spaces. By the previous lemmas, we know that all colors assigned to d_j has at least p items. That means besides U_l, there are at most $(k - 2)$ remaining colors on d_j. We modify the packing by assigning the remaining items of U_l to d_j. If the modification leads to a total load bigger than L, then we argue that there exists a large color $U_{l'} \neq U_l$ on d_j with more than p items. Otherwise, all remaining colors on d_j have no more than p items, a maximum of $k - 2$ such colors will never exceed load $(k - 2)p$, and together with the large color l will never exceed the load capacity. We simply truncate the number of items of $U_{l'}$ assigned to d_j, until either the total load on d_j is exactly L, or the items of $U_{l'}$ assigned to d_j is exactly p. In the latter case, we find another large color $U_{l''}$ with more than p items on d_j and so on. When we terminate, we have assigned U_l completely to d_j, no small colors are split, the number of unassigned small colors remains the same, no large color is partially assigned to a disk with number of items between $(0, p)$. So we have created a packing with one less large color partially

assigned to a disk with number of items between $(kq - kp + p, kq - kp + 2p)$, a contradiction to the choice of the initial packing. □

When a large color U_l is partially assigned to a disk d_j, we may view such assignment as splitting U_l into two new colors U_{l_1} and U_{l_2}, with U_{l_1} completely assigned to d_j. From the previous two lemmas, w.l.o.g., we may assume that in the optimal packing, in addition to the properties mentioned before, if a large color U_l is split for the first time, the resulting two new colors U_{l_1} and U_{l_2} each has at least p items.

LEMMA 3.10. *Consider the optimal packing $\mathcal{P}$ for an instance $\mathcal{I}$ with $A + a$ colors, for some integers $A, a > 0$. Let m_j denote the number of different colors assigned to disk d_j by $\mathcal{P}$. If $\sum_{i=1}^{N} m_j \leq A$, consider the modified instance $\mathcal{I}'$ with the A largest colors from $\mathcal{I}$. Then the optimal packing $\mathcal{P}'$ for I' has the same value as $\mathcal{P}$.*

PROOF. It is obvious that $\mathcal{P}'$ has value no more than $\mathcal{P}$. Consider $\mathcal{P}$, there are no more than A original colors assigned in $\mathcal{P}$. We could easily substitute the colors included in $\mathcal{P}$ with the A largest colors from $\mathcal{I}$, without changing the value of $\mathcal{P}$, thus creating a valid packing solution for $\mathcal{I}'$. This implies $\mathcal{P}'$ has at least the same value of $\mathcal{P}$. □

Given an optimal packing $\mathcal{P}$ for the created instance with the desired properties, we count the number of large colors that are not split. Let u denote such a number. So there are exactly $kq - kp - u$ split large colors. We consider an intermediate step of packing $\mathcal{P}$, where each split large color is split exactly once. We have argued that the resulting two large colors of any split large color has at least p items, so now we have an instance with $kN + (kq - kp - u)$ colors, each with at least p items. We claim that each unsplit large color reduces the storage capacity of the system by at least one. If a disk contains t large unsplit colors, then the remaining storage space is seemingly $k - t$, while the remaining load space is $kq - t(kq - kp + 2p) = (k - 2)p - (t - 1)(kq - kp + 2p) < (k - 2)p - (t - 1)2p = (k - 2t)p$. From Lemma 3.7 and 3.8, we know each color assigned to d_j has at least p items, thus d_j has no more than $k - 2t + t = k - t$ colors in $\mathcal{P}$. Overall, with u unsplit large colors, $\sum_{i=1}^{N} m_j \leq kN - u$. By the previous lemma, in the optimal packing $\mathcal{P}$, we must discard at least $kN + (kq - kp - u) - (kN - u) = kq - kp$ colors, each with at least p items. Thus the total number of items packed in $\mathcal{P}$ is at most $LN - (kq - kp)p = kq(kq - kp + p) - (kq - kp)p = kq(kq - kp) + kp^2$. This completes the discussion when k is not a perfect square.

3.2 Numerical Results

We have implemented the Sliding-Window algorithm and compared its performance to the theoretical results developed in this section. The details of this comparison are given in Appendix B. See Figs. 2–4 to see the plots of the fraction of unpacked items given by our worst case bound, the bound in [Shachnai and Tamir 2000a], as well as the simulation results.

3.3 Approximation Ratio

We have established the fact that the Sliding-Window algorithm packs at least $(1 - \frac{1}{(1+\sqrt{k})^2})$ fraction of all items, and this bound is the best possible among all possible algorithms by establishing instances where no solution can pack a higher fraction of items. Observe that the Sliding-Window algorithm also achieves a $(1 - \frac{1}{(1+\sqrt{k})^2})$-approximation ratio, since the optimal solution can pack all items. In this section we establish that the $(1-\frac{1}{(1+\sqrt{k})^2})$-approximation bound is tight by showing an example where the optimal packing packs all items while the Sliding-Window algorithm only packs $(1 - \frac{1}{(1+\sqrt{k})^2})$ fraction of all items.

For simplicity we show here the construction for the case when k is a perfect square. Let $L = (k + \sqrt{k})x$ and $N = \sqrt{k} + 1$, where x is a large integer to be determined later. We have $(k - \sqrt{k})(\sqrt{k} + 1)$ small colors, each have x items, and $\sqrt{k}(\sqrt{k} + 1)$ large colors of size $2x - 1$. The optimal packing puts $k - \sqrt{k}$ small colors and $\sqrt{k}$ large colors on each disk, for a total load of $(k + \sqrt{k})x - \sqrt{k}$ per disk, packing all items. Now consider the Sliding-Window algorithm, for the first disk d_1, it tries to find the first k consecutive colors exceeding L items. For the given instance, it will be $k - \sqrt{k} - 1$ small colors followed by $\sqrt{k} + 1$ large colors. The Sliding-Window algorithm then splits the last large color into two pieces of size $x + \sqrt{k}$ and $x - \sqrt{k} - 1$, respectively. The first disk is assigned $k - \sqrt{k} - 1$ small colors, $\sqrt{k}$ large colors, and a color of size $x + \sqrt{k}$, for a total of L items. Observe that the algorithm consumed $\sqrt{k} + 1$ large colors on the first disk d_1. It is clear that same situation arises for disks $d_2, d_3, \ldots, d_{\sqrt{k}}$. For the last disk, we can best assign k small colors to it, since all remaining colors have no more than x items each. Let $x \to \infty$, the ratio of packed (S) to total (S+U) items is

$$\frac{S}{S+U} = \frac{\sqrt{k}(k + \sqrt{k})x + kx}{(\sqrt{k}+1)((k + \sqrt{k})x - \sqrt{k})} \to \frac{\sqrt{k}(k + \sqrt{k})x + kx}{(\sqrt{k}+1)(k + \sqrt{k})x} = (1 - \frac{1}{(1 + \sqrt{k})^2}).$$

3.4 The Assignment Problem

Notice that the Sliding-Window algorithm returns the layout of the color placement, i.e., we know which color was assigned to which disk as well as a packing of items on disks. However, we could ignore the detailed item assignment, i.e., how many items from a color should be assigned to a disk. Observe that given the layout of the color placement only, one can find the optimal item assignment that maximizes the total fraction of the items packed via network flow. The reduction is as follows. We create a graph with the set of colors and disks as vertices, so a vertex represents either a color or a disk. In addition, we have a source and sink node, s and t, respectively. The source node s has an edge to a color node U_i with capacity $|U_i|$. From each disk node d_j there is an edge to t with capacity L_j. There is an edge with unlimited capacity from a color node U_i to a disk node d_j if in the Sliding-Window solution U_i is partially (or completely) assigned to d_j. Clearly, each disk node d_j has at most C_j edges from color nodes. A maximum flow from s to t gives an optimal assignment of items to disks.

However, we will show that the Sliding-Window algorithm itself returns an optimal assignment of items based on the color placement that it obtains. If all disks in the Sliding-Window solution are load saturated, then the observation is clearly

true.

THEOREM 3.11. *The Sliding-Window algorithm obtains an optimal packing of items for the layout it creates.*

PROOF. Since the sliding window algorithm finds an assignment of items to disks we can observe that it actually corresponds to a valid flow in the flow network described earlier. We can argue that this flow is actually a max flow in the flow network and thus optimal. Let f be the flow computed by the Sliding-Window algorithm and G_f the residual graph with respect to this flow.

The colors that are on the remaining items list when the algorithm terminates are the ones that have unpacked items. Call this set R_I. The corresponding color nodes U_i have the property that the edge from s to U_i is unsaturated and thus reachable from s. The disk nodes $d_j \in N_S$ (storage saturated disks) have edges to t that are not saturated and are the only nodes with edges to t in G_f. We will argue that there is no path from s to t in G_f, thus proving that the flow is a maximum flow. (Recall that paths in G_f correspond to augmenting paths in the flow network.)

Disks in N_S have the property that they do not cause splitting of any colors, since we pack all remaining items corresponding to a color. If a disk caused the splitting of a color it must be in N_L. We will show that each disk node d_j reachable from s in G_f caused the splitting of some color and thus must be in N_L. Since only disks in N_S have edges (in G_f) to t, there is no path in G_f from s to t.

We will prove, by induction on the length of the shortest path from s, that each disk node d_j reachable from s in G_f, caused the splitting of a color which is its predecessor on the shortest path from s. The base case is for all paths of length three from s. Any disk node d_j adjacent to a node in R_I caused the splitting of the node in R_I. This proves the base case. To prove the general case, suppose that the shortest path from s to d_j has length v. Let U_i be the predecessor of d_j on the shortest path and $d_{j'}$ the predecessor of U_i. By the induction hypothesis, $d_{j'}$ already has a predecessor that was split due to $d_{j'}$. Since each disk splits at most one color, and u_i has edges to $d_{j'}$ and d_j, it must be split due to d_j. This proves the claim. □

4. POLYNOMIAL TIME APPROXIMATION SCHEMES

4.1 Homogeneous Storage Systems

We now design an algorithm that for any fixed $\epsilon' > 0$, gives a $(1-\epsilon')$-approximation in polynomial time.

If the error parameter $\epsilon' \geq 1/(1+\sqrt{k})^2$, we can simply use the Sliding-Window algorithm to obtain a $(1-\epsilon')$-approximation. In the rest of this section, we focus on the case when $\epsilon' < 1/(1+\sqrt{k})^2$. In other words, k can be assumed to be a constant when ϵ' is a fixed constant. We also define a constant $\epsilon = \min(\frac{1}{k}, 1-(1-\epsilon')^{1/3})$. Our algorithm runs in polynomial time for any fixed k and ϵ and yields a $(1-\epsilon)^3 \geq (1-\epsilon')$ approximation. Our approximation scheme involves the following steps:

(1) First we show that any given input instance can be approximated by another instance I' such that no color in I' contains "too many" items.

(2) Next we show that for any input instance there exists a near-optimal solution that satisfies certain structural properties concerning how items are assigned to the disks.
(3) Finally, we give an algorithm that in polynomial time finds the near-optimal solution referred to in step (2) above, provided the input instance is as determined by step (1) above.

We now describe in detail each of these steps. In what follows, we use $\mathsf{OPT}(I)$ to denote an optimal solution to instance I and α to denote $1/\epsilon$. Also, for any solution S, we use $|S|$ to denote the number of items packed by it.

4.1.1 *Preprocessing the Input Instance.* We say that an instance I is *B-bounded* if the size of each color is at most B.

LEMMA 4.1. *For any instance I, we can construct in polynomial time another instance I' such that*

—*I' is $\lceil \alpha L \rceil$-bounded,*

—*any solution S' to I' can be mapped to a solution S to I of identical value, and*

—$|\mathsf{OPT}(I')| \geq (1-\epsilon)|\mathsf{OPT}(I)|$.

PROOF. Consider a color U_i in the instance I such that $|U_i| > \lceil \alpha L \rceil$. Replace U_i with a new set of colors $U_i^1, U_i^2, ..., U_i^s$ where $s = \lceil |U_i| / \lceil \alpha L \rceil \rceil$, $|U_i^1| = ... = |U_i^{s-1}| = \lceil \alpha L \rceil$, and $|U_i^s| = |U_i| - (s-1)\lceil \alpha L \rceil$. Repeat this procedure for any color that has more than $\lceil \alpha L \rceil$ items in I. We now have our instance I'.

It is easy to see that any feasible solution to I' gives a feasible solution of same value to I; simply replace each color U_i^j with U_i.

Now consider a solution S for instance I. We show that it can be mapped to a solution S' of size $(1-\epsilon)|S|$ for I'. If $|U_i| \leq \lceil \alpha L \rceil$ for $1 \leq i \leq M$, then clearly S is also a feasible solution of the same value for I'. Otherwise, fix a color U_i in I such that $|U_i| > \lceil \alpha L \rceil$. Label the occurrences of the items of color U_i as $1, 2, ...$ as we move from d_1 to d_N in solution S. Replace the jth occurrence of a color U_i item with an item of color U_i^l where $l = \lceil j / \lceil \alpha L \rceil \rceil$. The resulting solution may no longer be a feasible solution for I'. A disk may now contain items of two different colors, say U_i^l and U_i^{l+1}, in place of a single color U_i and hence the total number of colors in the disk may become $k+1$. We simply discard all the items in any disks where this event occurs. Repeat this procedure for every color with more than $\lceil \alpha L \rceil$ items in I. We claim that we have discarded no more than an ϵ-fraction of packed items. The reason is that we throw away at most L items from a color at a *crossover* disk but this event occurs only once in every $\lceil \alpha L \rceil$ occurrences of items packed from a color. Thus what we discard is at most an ϵ-fraction of what is packed. □

4.1.2 *Structured Approximate Solutions.* Let us call a color U_i *small* if $|U_i| \leq \epsilon L/k$, and *large* otherwise. Also, for a given solution, we say that a disk is *light* if it contains less than ϵL items, and it is called *heavy* otherwise. The lemma below shows that there exists a $(1-\epsilon)$-approximate solution where the interaction between light disks and large colors, and between heavy disks and small colors, obeys some nice properties.

LEMMA 4.2. *For any instance I, there exists a solution S satisfying the following properties:*

—*at most one light disk receives items from any large color.*

—*a heavy disk is assigned either zero or all items in a small color,*

—*a heavy disk receive no more than $\lceil (1-\epsilon)L \rceil$ items from large colors, and*

—*S packs at least $(1-\epsilon)\mathsf{OPT}(I)$ items.*

PROOF. Let n_j denote the number of items assigned to the jth disk in the solution $\mathsf{OPT}(I)$. Relabel disks 1 through N such that $n_1 \geq n_2 ... \geq n_N$. Assume w.l.o.g. that $\mathsf{OPT}(I)$ is a lexicographically maximal solution in the sense that among all optimal solutions, $\mathsf{OPT}(I)$ is one that maximizes the sums $\sum_{j=1}^{i} n_j$ for each $i \in [1..N]$.

It is easy to see that the first property follows from the maximal property of $\mathsf{OPT}(I)$. To establish that a heavy disk in $\mathsf{OPT}(I)$ receives either zero or all items from a small color in the solution S, we may need to discard some items from the heavy disks in $\mathsf{OPT}(I)$. Let X be the set of heavy disks that contain less than $\lceil (1-\epsilon)L \rceil$ items from large colors. Consider any disk $d_j \in X$ that receives some but not all items from a small color U_i. Simply move all items of U_i to d_j. Repeat this process till no disk in X violates this property. Since a small color has at most $\epsilon L/k$ items, clearly the capacity of no disk is violated in this process. Finally, for each of the remaining heavy disk not in X, simply discard any items from small colors and extra items from large colors, so the resulting load is exactly $\lceil (1-\epsilon)L \rceil$. Clearly, the resulting solution is $(1-\epsilon)$-approximate. □

For a given solution S, a disk is said to be δ-integral w.r.t. a color U_i if it is assigned $\beta\lceil \delta L \rceil$ items from U_i, where $0 < \delta \leq 1$ and β is a non-negative integer.

LEMMA 4.3. *Any solution S can be transformed into a solution S' such that*

—*each heavy disk in S is (ϵ^2/k)-integral in S' w.r.t. each large color, and*

—*S' packs at least $(1-\epsilon)|S|$ items.*

PROOF. To obtain the solution S' from S, in each heavy disk, round down the number of items assigned from any large color to the nearest integral multiple of $\lceil (\epsilon^2/k)L \rceil$. Then the total number of items discarded from any heavy disk in this process is at most

$$k\left(\left\lceil (\frac{\epsilon^2}{k})L \right\rceil - 1\right) \leq k\left((\frac{\epsilon^2}{k})L\right) = \epsilon(\epsilon L).$$

Since each heavy disk contains at least ϵL items, the total number of items discarded in this process can be bounded by $\epsilon|S|$. Thus S' satisfies both properties above. □

4.1.3 *The Approximation Scheme.* We start by preprocessing the given input instance I so as to create an $\lceil \alpha L \rceil$-bounded instance I' as described in Lemma 4.1. We now give an algorithm to find a solution S to I' such that S satisfies the properties described in Lemmas 4.2 and 4.3 and packs the largest number of items. Clearly,

$$|S| \geq (1-\epsilon)^2|\mathsf{OPT}(I')| \geq (1-\epsilon)^3|\mathsf{OPT}(I)|.$$

Let O be an optimal solution to the instance I' that is lexicographically maximal. Assume w.l.o.g. that we know the number of heavy disks in O, say N'. Let $\mathcal{H}$ be the set of heavy disks d_1 through $d_{N'}$ and let $\mathcal{L}$ be the remaining disks, $d_{N'+1}$ through d_N. The algorithm consists of two steps, corresponding to the packing of items in $\mathcal{H}$ and $\mathcal{L}$ respectively.

4.1.3.1 *Packing items in $\mathcal{H}$:.* We first guess a vector $\langle l_1, l_2, ..., l_{N'}\rangle$ such that l_j denotes the number of small colors to be assigned (completely) to a disk $d_j \in \mathcal{H}$. Since all disks are identical, we can guess such a vector in $O(N^{k+1})$ time by guessing a compact representation of the following form. We guess a vector $\langle q_0, q_1, \cdots, q_k\rangle$ such that $\sum_{i=0}^{k} q_i = N'$ and q_i denotes the number of disks in $\mathcal{H}$ that are assigned i small colors (completely). It is easily seen that any such vector can be mapped to a vector of the form $\langle l_1, l_2, ..., l_{N'}\rangle$ and vice versa. Now proceeding from 1 through N', we assign to a disk d_j the largest size l_j small colors that remain.

Next we use a dynamic programming approach moving across the disks from d_1 through $d_{N'}$ so as to find an optimal (ϵ^2/k)-integral solution for packing the largest number of items from the large colors. For the purpose of this packing, the capacity of each heavy disk is restricted to be $\lceil(1-\epsilon)L\rceil$ and the number of colors allowed in disk d_j is given by $k - l_j$. Let $\beta = \lceil k/\epsilon^3\rceil$ and $q = \lceil(\epsilon^2 L)/k\rceil$. The dynamic program is based on maintaining a β-tuple $\vec{v} = \langle v_1, v_2, ..., v_\beta\rangle$ where v_i denotes the number of large colors that have $i \cdot q$ elements available in them. Proceeding from $i = 1$ through N', we compute a table entry $T[\vec{v}, i]$ for each possible state vector $\vec{v}$. The entry indicates the largest number of items that can be packed in the disks d_1 through d_i subject to the constraint that the resulting state vector is $\vec{v}$. Since there are at most Nk colors, total number of state vectors is bounded by $(Nk)^{\lceil k/\epsilon^3\rceil}$, which is polynomial for any fixed ϵ.

4.1.3.2 *Packing items in $\mathcal{L}$:.* We know that our solution need not assign items from a large color to more than one disk in $\mathcal{L}$. Moreover, at most ϵL items from any large color are packed in a disk in $\mathcal{L}$. So at this stage we can truncate down the size of each large color to ϵL. By definition, each of the remaining colors has no more than $\frac{L}{k}$ items, any k of them will underfill a disk. We greedily assign the colors with the most items to disks in $\mathcal{L}$, so each disk is storage saturated. Clearly, such an assignment gives a feasible solution of maximum weight. This completes our approximation scheme.

4.2 Uniform Ratio Storage Systems

We now show how the PTAS above can be extended to the case of uniform ratio storage systems. If $\epsilon' \geq 1/(1+\sqrt{C_{\min}})^2$ then we can use the Sliding-Window algorithm to obtain a $(1-\epsilon')$-approximation. On the other hand, if $C_{\min}$ as well as $C_{\max}$ are bounded by a constant (parameterized by ϵ'), the approach of the preceding subsection easily extends to give a PTAS. We will present this extension later. Define ϵ as before, using $C_{\max}$ in place of k.

The difficulty thus lies in the case when $C_{\min}$ is small but $C_{\max}$ is relatively large. In other words, our system contains disks of widely varying storage capacities. We handle this case by showing that every "large" disk can be approximately represented by a collection of disks with constant storage capacity such that we

lose at most an ϵ-fraction of items due to this approximate representation. Once this transformation is made, we again have a case where both C_{min} and C_{max} are bounded by a constant.

LEMMA 4.4. *For any instance I, we can construct in polynomial time another instance I' such that*

—*Each disk in I' has bounded constant storage capacity.*

—*Any solution S' to I' can be mapped to a solution S to I of identical value, and*

—$|\mathsf{OPT}(I')| \geq (1-\epsilon)|\mathsf{OPT}(I)|$.

PROOF. We create the instance I' as follows. Let p be the smallest integer such that $\epsilon \geq 1/(1+\sqrt{p \times C_{\min}})^2$. For each disk d_j with storage capacity $C_j > 2p \times C_{\min}$, we decompose it into a collection of $\lfloor \frac{C_j}{pC_{\min}} \rfloor - 1$ disks of storage capacity $p \times C_{\min}$ and one more disk of storage capacity $(C_j \mod pC_{\min}) + pC_{\min}$. We ensure that disks have the same uniform ratio. Note that in the new instance, $C_{max} \leq 2p \times C_{\min}$, which is a constant.

It's easy to see that any feasible solution to I' gives a feasible solution of same value to I, since the total storage and load capacity remains the same.

Now consider a solution S for I. For each of the disks d_j in I with storage capacity larger than $2p \times C_{\min}$, we set up a URDP instance with the colors assigned to d_j as the set of items, and the collection of the disks representing d_j in I' as the set of disks. Since the minimum storage capacity of the disks in the collection is $pC_{\min}$, we can run the Sliding-Window algorithm to pack at least $(1-\epsilon)$ fraction of the items. □

We now present a PTAS for the uniform ratio system with bounded storage capacities. It involves three steps as in the PTAS for the homogeneous system. Recall $r = \frac{L_j}{C_j}$ denotes the uniform ratio and $\alpha = \frac{1}{\epsilon}$. We inherit the notations and concepts introduced in the previous subsection.

4.2.1 *Preprocessing the Input Instance.* We preprocess the instance so no color contains "too many" items.

LEMMA 4.5. *For any instance I, we can construct in polynomial time another instance I' such that*

—*I' is $\lceil \alpha L_{\max} \rceil$-bounded.*

—*any solution S' to I' can be mapped to a solution S to I of identical value, and*

—$|\mathsf{OPT}(I')| \geq (1-\epsilon)|\mathsf{OPT}(I)|$.

PROOF. The same proof in Lemma 4.1 works here. In the case that we throw away some items at a *crossover* disk, at most $L_{\max}$ items are lost. But this event occurs only once in every $\lceil \alpha L_{max} \rceil$ occurrences of items packed from a color. Thus what we discard is at most an ϵ-fraction of what is packed. □

4.2.2 *Structured Approximate Solutions.* Let us call a color U_i *small* if $|U_i| \leq \epsilon r$, and *large* otherwise. For a given solution, we say that a disk d_j is *light* if it contains less than ϵL_j items, and it is called *heavy* otherwise. The lemma below is an analog of Lemma 4.2.

LEMMA 4.6. *For any instance I, there exists a solution S satisfying the following properties:*

—*at most one light disk receives items from any large color,*
—*a heavy disk is assigned either zero or all items in a small color,*
—*a heavy disk d_j receive no more than $\lceil(1-\epsilon)L_j\rceil$ items from large colors, and*
—*S packs at least $(1-\epsilon)\mathsf{OPT}(I)$ items.*

PROOF. The proof mimics that of lemma 4.2. We relabel the disks in decreasing order of number of assigned items. A lexicographically maximal solution $\mathsf{OPT}(I)$ satisfies the first property. Let X denote the set of disks with each of its members d_j containing less than $\lceil(1-\epsilon)L_j\rceil$ items from large colors. We completely assign the partially assigned small colors to disks in X. Afterwards, each disk $d_j \in X$ contains no more than C_j small colors. Since a small color has at most ϵr items, d_j contains at most ϵL_j items from small colors. Together with less than $\lceil(1-\epsilon)L_j\rceil$ items from large colors, the load capacity of d_j is not violated. For each of the remaining heavy disk d_j not in X, simply discard any items from small colors and extra items from large colors, so the resulting load is exactly $\lceil(1-\epsilon)L_j\rceil$. Clearly, the resulting solution is $(1-\epsilon)$-approximate. □

For a given solution S, a disk is said to be δ-integral w.r.t. a color U_i if it is assigned $\beta\lceil\delta r\rceil$ items from U_i, where $0<\delta\leq 1$ and β is a non-negative integer. Note that the definition here is different from the previous subsection, though achieves the same effect.

LEMMA 4.7. *Any solution S can be transformed into a solution S' such that*

—*each heavy disk in S is ϵ^2-integral in S' w.r.t. each large color, and*
—*S' packs at least $(1-\epsilon)|S|$ items.*

PROOF. To obtain the solution S' from S, in each heavy disk d_j, round down the number of items assigned from any large color to the nearest integral multiple of $\lceil\epsilon^2 r\rceil$. Then the total number of items discarded from d_j in this process is at most

$$C_j\left(\lceil\epsilon^2 r\rceil - 1\right) \leq C_j\left(\epsilon^2 r\right) = \epsilon(\epsilon L_j).$$

Since d_j contains at least ϵL_j items, the total number of items discarded in this process can be bounded by $\epsilon|S|$. Thus S' satisfies both properties above. □

4.2.3 *The Approximation Scheme.* We start by preprocessing the given input instance I so as to create an $\lceil\alpha L_{max}\rceil$-bounded instance I' as described in Lemma 4.5. We now give an algorithm to find a solution S to I' such that S satisfies the properties described in Lemmas 4.6 and 4.7 and packs the largest number of items. Clearly,

$$|S| \geq (1-\epsilon)^2|\mathsf{OPT}(I')| \geq (1-\epsilon)^3|\mathsf{OPT}(I)|.$$

Let O be an optimal solution to the instance I' that is lexicographically maximal. Assume w.l.o.g. that we know the number of heavy disks in O, say N'. For the uniform ratio system, we can assume all disks are ordered in decreasing order of

their storage capacities in the lexicographically maximal solution, and disks d_1 through $d_{N'}$ are heavy disks (denoted by the set $\mathcal{H}$) and the rest are light disks (denoted by the set $\mathcal{L}$).

Packing items in $\mathcal{H}$: We first guess a vector $\langle l_1, l_2, ..., l_{N'}\rangle$ such that l_j denotes the number of small colors to be assigned completely to a disk $d_j \in \mathcal{H}$. Since there are constant number of disk types, we can guess such a vector in $O(N^{O(C^2_{max})})$ time by guessing a compact representation of the following form. We guess a vector $\langle Q_0^{C_{\min}}, Q_1^{C_{\min}}, \cdots, Q_{C_{\max}}^{C_{\max}}\rangle$ such that $\sum Q_i^j = N'$ and Q_i^j denotes the number of disks with storage capacity j in $\mathcal{H}$ that are assigned i small colors (completely). It is easily seen that any such vector can be mapped to a vector of the form $\langle l_1, l_2, ..., l_{N'}\rangle$ and vice versa. Now proceeding from 1 through N', we assign to a disk d_i the largest size l_i small colors that remain.

Next we use the same dynamic program described in the previous section to find an optimal ϵ^2-integral solution for packing the largest number of items from the large colors. For the purpose of this packing, the capacity of each heavy disk d_j is restricted to be $\lceil(1-\epsilon)L_j\rceil$ and the number of colors allowed is given by $C_j - l_j$.

Packing items in $\mathcal{L}$: We have argued that each large color is assigned to at most one light disk. So at this stage we can truncate down the size of each large color to no more than $\epsilon L_{\max}$. By definition, $\epsilon \leq \frac{1}{C_{\max}}$, which suffices for the condition $\epsilon L_{\max} \leq \frac{L_{\max}}{C_{\max}} = r$. Now the remaining colors each has no more than r items, any C_j of them will underfill the disk d_j. We greedily assign the colors with the most items to disks in $\mathcal{L}$, so each disk is storage saturated. Clearly, such an assignment gives a feasible solution of maximum weight. This completes our approximation scheme for the uniform ratio systems.

REFERENCES

Berson, S., Ghandeharizadeh, S., Muntz, R. R., and Ju, X. 1994. Staggered Striping in Multimedia Information Systems. *ACM SIGMOD Conference*, 79–90.

Chekuri, C. and Khanna, S. 1999. On multidimensional packing problems. In *ACM/SIAM Symp. on Discrete Algorithms.* 185–194.

Chervenak, A. L. 1994. Tertiary Storage: An Evaluation of New Applications. *Ph.D. Thesis, UC Berkeley.*

Chou, C.-F., Golubchik, L., Lui, J., and Chung, I.-H. 2002. Design of Scalable Continuous Media Servers. *Special issue on QoS of Multimedia Tools and Applications 17,* 2-3, 181–212.

Cormen, T. H., Leiserson, C. E., and Rivest, R. L. 1990. *Introduction to Algorithms.* The MIT Press and McGraw-Hill Book Company.

Frieze, A. M. and Clarke, M. R. B. 1984. Approximation algorithms for the m-dimensional 0-1 knapsack problem: worst-case and probabilistic analyses. *European Journal of Operational Research*, 100–109.

Garey, M. R. and Johnson, D. S. 1979. *Computers and intractability: A guide to the theory of NP-completeness.* Freeman, San Francisco.

Kashyap, S. and Khuller, S. 2003. Algorithms for non-uniform size data placement on parallel disks. In *Foundations of Software Technology and Theoretical Computer Science (FST&TCS).* 265–276.

Knuth, D. E. 1973. *The Art of Computer Programming, Volume 3.* Addison-Wesley.

Pugh, W. 1990. Skip lists: A probabilistic alternative to balanced trees. *Communications of the ACM 33,* 6, 668–676.

Raghavan, P. 1988. Probabilistic construction of deterministic algorithms: approximating packing integer programs. *Journal of Computer and System Sciences*, 130–143.

Shachnai, H. and Tamir, T. 2000a. On two class-constrained versions of the multiple knapsack problem. *Algorithmica 29,* 3, 442–467.

Shachnai, H. and Tamir, T. 2000b. Polynomial time approximation schemes for class-constrained packing problems. In *Workshop on Approximation Algorithms (APPROX)*. 238–249.

Shachnai, H. and Tamir, T. 2003. Approximation schemes for generalized 2-dimensional vector packing with application to data placement. In *Workshop on Approximation Algorithms (APPROX)*. 238–249.

Stonebraker, M. 1986. A Case for Shared Nothing. *Database Engineering 9,* 1, 4–9.

Wolf, J., Yu, P., and Shachnai, H. 1995. DASD dancing: A disk load balancing optimization scheme for video-on-demand computer systems. In *Proceedings of the 1995 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*. 157–166.

A. NP-HARDNESS PROOF

In this appendix we give our proof of Theorem 1.3. We first define the PARTITION problem, and then describe a polynomial time reduction from it to the homogeneous data placement problem.

PARTITION: Given a finite set A, with a size $s(a)$ for each element $a \in A$, is there a subset $A' \subseteq A$ such that

$$\sum_{a \in A'} s(a) = \sum_{a \in A - A'} s(a)?$$

This remains NP-complete even if we require $|A'| = |A|/2$ [Garey and Johnson 1979]. Let $n = |A|$.

Proof. The problem is easily seen to be in NP, since a proposed solution can be trivially checked in polynomial time.

We now show a polynomial time reduction from PARTITION. Let $a_{\max}$ be the maximum size item. For the reduction, define $K = n \times C \times s(a_{\max})$ where C is a large constant. Let $L = \frac{3}{2}K$ and $N = n$. So there are n disks with $k = 2$ (storage capacity) and a large L value. Let $D = \sum_{i=1}^{n}(K + s(a_i)) = n \times K + \sum_{i=1}^{n} s(a_i)$. Let $M = n + 2$, with $x_i = \frac{1}{2}K - s(a_i)$ for $1 \le i \le n$ and $x_{n+1} = \frac{1}{2}D$ and $x_{n+2} = \frac{1}{2}D$.

Note that $\sum_{i=1}^{n+2} x_i = \sum_{i=1}^{n} x_i + D = \frac{3}{2}n \times K$ which is exactly NL, the storage capacity.

We now claim that if there is a solution to the partition problem with $|A'| = |A|/2$ and with $s(A') = s(A - A')$ then there is a way to pack all items into the N disks. The items are packed as follows. Put x_i items of color i in disk i. Disk i now has space for one new color, and exactly $\frac{3}{2}K - (\frac{1}{2}K - s(a_i))$ items. This is exactly $K + s(a_i)$. If item $a_i \in A'$ then add items of color $n + 1$ to disk i, otherwise add items of color $n + 2$ to disk i. Since each disk can hold two colors, this does not violate the color. Note that the number of items of color $n + 1$ that we pack are exactly $\sum_{a_i \in A'}(K + s(a_i)) = \frac{n}{2} \times K + s(A') = \frac{1}{2}D$. The calculation is identical for items of color $n + 2$, and this concludes the proof that all items are packed.

We now argue that if there is a solution to 2-HDP where all items get packed, then there is a solution to the PARTITION problem. We first claim that if all items are packed, then items of colors i and j, with $1 \le i, j \le n$ cannot be packed into the same disk. This is the case since: (a) only two colors can be packed in a disk and hence no other color can go into that disk, and (b) the total capacity used up by items of color i and j, $1 \le i, j \le n$, would not exceed K, which is much less than the capacity of the disk. If we cannot saturate the disk to full load capacity, then we cannot pack all items (since the number of items exactly equals the total load capacity). If each items of color i is in a distinct disk, then without loss of generality we pack items of color i in disk i and now we are left with items of only two colors that we need to split equally between the disks, and each disk can only take an item of one color, with $K + s(a_i)$ items of that color. Since $K >> s(a_i)$ we must pack items of color $n + 1$ in exactly $n/2$ disks, and the items of color $n + 2$ in the remaining $n/2$ disks. Let $A' = \{a_i | \text{items of color } n + 1 \text{ are packed in disk } i\}$. As said earlier $|A'| = \frac{n}{2}$, and $s(A') = s(A - A')$. This completes the proof. □

When $k = 3$, the problem can be seen to be strongly NP-hard for even the homogenous case by a simple reduction from 3-PARTITION [Garey and Johnson 1979].

B. THE SLIDING-WINDOW ALGORITHM

Algorithm 1 shows the pseudo code of the Sliding-Window algorithm. $HEAD$ and $TAIL$ always point to the first and last color in the list, respectively. We'll actively update $LEFT$ to points to the first (leftmost) color in the window, according to the current sorted color list. Likewise $RIGHT$ points to the last (rightmost) color in the window. In addition, $WIDTH$ denotes the total number of colors in the currently window, and $iter$ is a temporary pointer variable. $\mathcal{PREV}(p)$ and $\mathcal{NEXT}(p)$ are pointer operations that returns a pointer to the color before and after the color pointed to by p. We use the convention that $\mathcal{PREV}(HEAD) = nil$, and $\mathcal{NEXT}(nil) = HEAD$. A sliding window for disk d_j contains no more than C_j consecutive colors. SUM denotes the total number of items of all the colors in the current window.

```
   iter = nil
   foreach disk d_j, j = 1...N do
       RIGHT = iter
       SUM = WIDTH = 0
1      while iter ≠ nil and WIDTH < C_j do                /* Initial window */
           SUM+ = R[iter];  ++WIDTH;  iter = PREV(iter)
       LEFT = NEXT(iter)
2      while SUM < L_j and RIGHT ≠ TAIL do                /* Sliding window */
           RIGHT = NEXT(RIGHT);  SUM+ = R[RIGHT]
           if WIDTH == C_j then
               SUM− = R[LEFT];  LEFT = NEXT(LEFT)
           else
               WIDTH++
3      iter = PREV(LEFT)          /* Initial RIGHT value for next disk */
       Assign items R[LEFT]...R[RIGHT−1] to disk d_j & delete them from R.
       if SUM > L_j then                          /* Split color if necessary */
           Assign items R[RIGHT] − SUM + L_j to disk d_j and delete
           R[RIGHT].
           Creating a new color with SUM − L_j items and Insert into R.
       else
           Assign items R[RIGHT] to disk d_j and delete it from R.
```

Algorithm 1: SLIDING-WINDOW

The reason for the improvement over the original $O(NM)$ running time is that we do not have to start the window of length C_j, from the beginning of the list R in each step j. We count the colors split as new types of colors. Thus the total number of colors is upper bounded by $O(N + M)$.

For the **while** loop noted by 1, the number of iterations is upper bounded by the total number of colors in the final window, which are deleted from the list later.

Once a color is deleted from the list, it does not interact with the algorithm any more. Thus, the total number of executions of the loop is bounded by $O(N+M)$ as well.

We then count the total number of "window slides" (the **while** loop noted by 2). Consider step $j-1$, we have determined the colors $R[LEFT]\ldots R[RIGHT]$ to be placed in knapsack d_{j-1}. Now *iter* points to the color previous to that of $LEFT$ (noted by 3). Since the colors are sorted, the window ending with $R[iter]$ will under-fill disk d_j. That's why the window ending with the color pointed to by *iter* should be a safe starting position for the next iteration. Thus in subsequent "window slides" during step j, the right end of the window extends to a new color per slide. Thus, the right end of the window will always monotonically move to the right, consuming a fresh color in each step. Since the total number of colors is bounded by $O(N+M)$, this upper bounds the total number of "window slides" in the pseudo code.

To achieve an overall $O((N+M)\log(N+M))$ running time, we need a data structure that maintains a sorted list of colors, supporting bidirectional traversals of the list in constant time, plus deletion or addition of a single color in logarithmic time. One can adapt a balanced binary search tree type data structure to support the needed operation. For example, a Skip-list [Pugh 1990] or a 2-3 tree [Cormen et al. 1990] together with a doubly linked list would satisfy the requirement.

Numerical Results

In this appendix we give numerical results of the comparison between the Sliding-Window algorithm and the theoretical results developed in Section 3. The results presented here are for the *homogeneous* case only. For the purposes of this comparison we generate the test cases using the Zipf distribution to determine the skewness in the number of items of each color. The Zipf distribution is defined as follows [Knuth 1973]:

$$\text{Prob}[\text{item of color } i] = \frac{c}{i^{(1-\theta)}} \qquad \forall\ i = 1,2,\ldots,M \quad \text{and} \quad 0 \le \theta \le 1$$

$$\text{where} \quad c = \frac{1}{H_M^{(1-\theta)}} \qquad \text{and} \qquad H_M^{(1-\theta)} = \sum_{j=1}^{M} \frac{1}{j^{(1-\theta)}}$$

where θ determines the degree of skewness. For instance, $\theta = 1.0$ corresponds to the uniform distribution whereas $\theta = 0.0$ corresponds to the *measurements* performed in [Chervenak 1994] (for a movies-on-demand application).

We experimented with different values of θ and computed the percentage of items that can be packed by the Sliding-Window algorithm as a function of k, the load capacity of a disk. The results of these experiments are given in Figures 2–4. In all cases $L = 100$ and $N = 5$.

We can draw the following conclusions from these figures:

—the theoretical bound is reasonably tight when the the number of items of each color is fairly skewed (as in Figures 3 and 4), as is the case in a VOD server, which is our motivational application; furthermore, the performance of the Sliding-Window algorithm is very close to the theoretical bound for inputs which are "similar" to the tight example given in Section 3 (as in Figure 3);

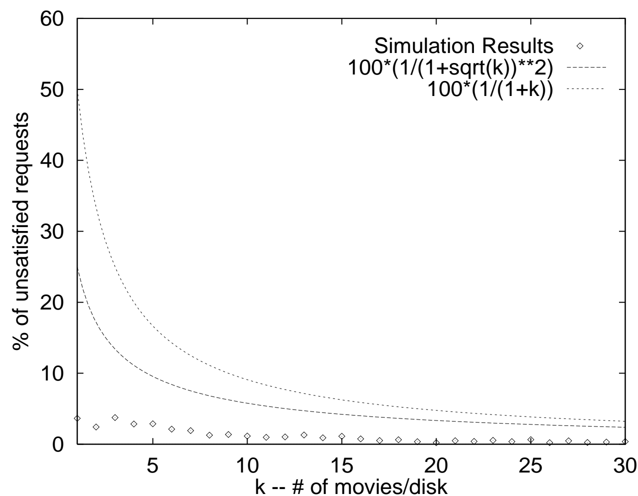


Fig. 2. $\theta = 1.0$.

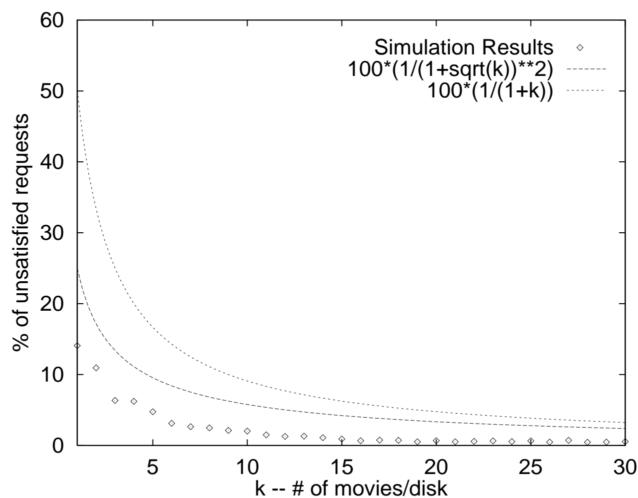


Fig. 3. $\theta = 0.5$.

—the performance of the Sliding-Window algorithm can be significantly better than the theoretical bound when the number of items of each color is approximately the same and each disk has a relatively small storage capacity (as in Figure 2); however, the theoretical bound is reasonably tight for larger values of k (which again, is reasonable for our motivational application).

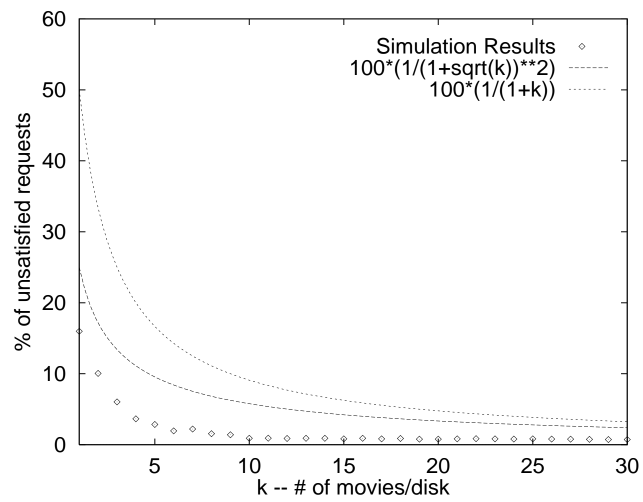


Fig. 4. $\theta = 0.0$.