Algorithms for Non-Uniform Size Data Placement on Parallel Disks

Srinivas Kashyap *   Samir Khuller †

Abstract

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 items) 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 item. Data item i has size s_i. Each disk d_j is characterized by two parameters, namely, its *storage capacity* C_j which indicates the maximum total size of data items 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 items 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 *homogeneous* storage systems where all the disks are identical. We assume that all disks have a storage capacity of k and a load capacity of L. Previous work on this problem has assumed that all data items have *unit* size, in other words $s_i = 1$ for all i. Even for this case, the problem is NP-hard. We present the first successful attempt at removing the assumption that all data items have the same size. For the case where $s_i \in \{1, \ldots, \Delta\}$ for some constant Δ, we develop a polynomial time approximation scheme (PTAS). This result is obtained by developing two algorithms, one that works for constant k and one that works for arbitrary k. The algorithm for arbitrary k guarantees that a solution where at least $\frac{k-\Delta}{k+\Delta}\left(1 - \frac{1}{\left(1+\sqrt{\frac{k}{2\Delta}}\right)^2}\right)$ fraction of all clients are assigned to a disk. In addition we develop an algorithm for which we can prove tight bounds when $s_i \in \{1, 2\}$. In particular, we can show that a $(1 - \frac{1}{(1+\sqrt{\lfloor k/2 \rfloor})^2})$ fraction of all clients can be assigned, regardless of the input distribution.

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 items) 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 item. Each data item has size s_i. Each disk d_j is characterized by two parameters, namely, its *storage capacity* C_j which indicates the maximum storage capacity for data items that may be placed on 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 items 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 items such as movies on a system of disks. The main difference between this type of data access problem and traditional data access problems are that in this situation, once assigned, the clients will receive multimedia data continuously and will not be queued. Hence we would like to maximize the number of clients that can be assigned/admitted to the system. We study this data placement problem for uniform storage systems, or a set of identical disks where $C_j = k$ and $L_j = L$ for all disks d_j.

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., $\sum_{i=1}^{M} |U_i| \leq N \cdot L$, and (ii) the total size of data items does not exceed the total storage capacity, i.e., $\sum_{i=1}^{M} s_i \leq N \cdot k$ and (iii) If M_p is the number of data items of size p then $M_p \leq N\lfloor \frac{k}{p} \rfloor$, since at most $\lfloor \frac{k}{p} \rfloor$ items of size p can be stored on a single disk.

In [4, 8] this problem is studied with the assumption that all data items have unit size, namely $s_i = 1$ for all data items, and even this case is NP-hard for homogeneous disk systems [4]. In this work, we generalize

*Department of Computer Science University of Maryland, College Park, MD 20742. Research supported by a Senior Summer Scholar Award and NSF Award CCR-0113192. E-mail : raaghav@cs.umd.edu.

†Department of Computer Science and Institute for Advanced Computer Studies, University of Maryland, College Park, MD 20742. Research supported by NSF Award CCR-9820965 and CCR-0113192. E-mail : samir@cs.umd.edu.

this problem to the case where we can have non-uniform sized data items. For the previous algorithms [4, 8] the assumption that all items have the same size is crucial.

For arbitrary k and when $s_i \in \{1, 2\}$ (this corresponds to the situation when we have two kinds of movies - standard and large), we develop a generalization of the sliding-window algorithm [8], called SW-Alg, using multiple lists, that has the following property. For any input distribution that satisfies the size requirements mentioned above, we can show that the algorithm guarantees that at least $(1 - \frac{1}{(1+\sqrt{\lfloor k/2 \rfloor})^2})$ fraction of the clients can be assigned to a disk. Note that $(1 - \frac{1}{(1+\sqrt{\lfloor k/2 \rfloor})^2})$ approaches 1 as k increases, and is at least $\frac{3}{4}$. This bound holds for $k \geq 2$. While this bound is trivial when k is even, the proof is quite complicated for odd k. *In addition, we show that this bound is tight.* In other words there are instances where no placement of data items can guarantee a better bound as a function of k. In fact, this suggests that when $s_i \in \{1, \ldots, \Delta\}$ we should get a bound of $(1 - \frac{1}{(1+\sqrt{\lfloor k/\Delta \rfloor})^2})$. We have verified this conjecture when $\Delta = 2$. (We can argue that this is the best bound possible, by using the same approach as in Appendix B.)

For the more general problem when $s_i \in \{1, \ldots, \Delta\}$. we develop a new method (SW-Alg2) that works with a single list of all the items, sorted in non-decreasing density order. This algorithm has the property that at least $f(k, \Delta) = \frac{k-\Delta}{k+\Delta} \left(1 - \frac{1}{\left(1+\sqrt{\frac{k}{2\Delta}}\right)^2}\right)$ fraction of all clients are assigned. When $s_i \in \{1, \ldots, \Delta\}$ for some constant Δ, we develop a polynomial time approximation scheme (PTAS) as follows. For a given $\epsilon > 0$, if $(1-\epsilon) \leq f(k, \Delta)$ then we can use SW-Alg2 to get the desired result. If $(1 - \epsilon) > f(k, \Delta)$, then k is a fixed constant (as a function of ϵ and Δ) and we can use an algorithm whose running time is polynomial for fixed k. In fact, this algorithm works when $s_i \in \{a_1, \ldots, a_c\}$ for any fixed constant c. This generalizes the algorithm presented in [4], which is for the case when all $s_i = 1$. While the high level approach is the same, the algorithm is significantly more complex in dealing with lightly loaded disks. For any fixed integer k, Δ and $\epsilon > 0$ this algorithm runs in polynomial time and outputs a solution where at least $(1 - \epsilon)$ fraction of the clients are assigned.

1.1 Related Work. The data placement problem described above bears some resemblance to the classical multi-dimensional knapsack problem [6, 2]. However, in our problem, the storage dimension of a disk behaves in a *non-aggregating* manner in that assigning additional clients corresponding to a data item that is already present on the disk 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 [8] studied the above data placement problem for unit sized data items when all $s_i = 1$; 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 $\sum_{j=1}^{N} C_j \geq M + N - 1$. An easy corollary of this result is that one can always pack a $(1 - \frac{1}{1+k})$-fraction of all items. The authors [8] showed that the problem is NP-hard when each disk has an arbitrary load capacity, and unit storage. Golubchik et. al. [4] establish a tight upper and lower bound on the number of items that can *always* be packed for any input instance to homogeneous storage systems, regardless of the distribution of requests for data items. It is always possible to pack a $(1 - \frac{1}{(1+\sqrt{k})^2})$-fraction of items for any instance of identical disks. Moreover, there exists a family of instances for which it is infeasible to pack any larger fraction of items. The problem with identical disks is shown to be NP-hard for any fixed $k \geq 2$ [4].

In addition, packing problems with color constraints are studied in [3, 7]. Here items have sizes and colors, and items have to be packed in bins, with the objective of minimizing the number of bins used. In addition, each item has a color and there is a constraint on the number of items of distinct colors in a bin. For a constant total number of colors, the authors develop a polynomial time approximation scheme. In our application, this translates to a constant number of data items (M), and is too restrictive an assumption.

1.2 Other Issues. Once a placement of items on the disks has been obtained, the problem of assigning clients to disks can be solved optimally by solving a network flow instance. Our algorithm computes a data placement and an assignment, however it is possible that a better assignment can be obtained for the same placement by solving the appropriate flow problem. (For the unit size case this is not an issue since we can show that the assignment is optimal for the placement that is produced by the sliding window algorithm.)

Another important issue concerns the input size of the problem. The input parameters are N, the number of disks, and $M(\leq Nk)$ the total number of movies. Since only the cardinalities of the sets U_i are required, we assume each of these can be specified in $O(\log |U_i|)$ bits. In other words, our algorithms run in time polynomial in these parameters and are not affected by exceptionally large sets U_i, assuming we can manipulate these values in constant time.

In Appendix A we describe in some detail the motivating application for our data placement problem.

1.3 Main Results. When data items have size $s_i \in \{1, 2\}$, we develop a generalization of the Sliding Window Algorithm (SW-Alg) using multiple lists, and prove that it guarantees that at least $(1 - \frac{1}{(1+\sqrt{\lfloor k/2 \rfloor})^2})$ fraction of clients will be assigned to a disk. Note that this function is always at least $\frac{3}{4}$ and approaches 1 as k goes to ∞. Moreover, we can show that this bound is tight. In other words there are client distributions for which no layout would give a better bound. Developing tight bounds for this problem turn out to be quite tricky, and much more complex than the case where all items have unit size. This already allows for understanding the fragmentation effects due to imbalanced load as well as due to non-uniform item sizes. We were able to develop several generalizations of the sliding window method, but it is hard to prove tight bounds on their behavior.

In addition, we develop a new algorithm (SW-Alg2) for which we can prove that it guarantees that at least $f(k, \Delta) = \frac{k-\Delta}{k+\Delta}\left(1 - \frac{1}{\left(1+\sqrt{\frac{k}{2\Delta}}\right)^2}\right)$ fraction of clients will be assigned to a disk, when $s_i \in \{1, \dots, \Delta\}$.

As mentioned earlier, by combining SW-Alg2 with an algorithm that runs in polynomial time for fixed k we can obtain a polynomial time approximation scheme. We develop an algorithm (Apx-Scheme) that takes as input parameter two constants k and ϵ' and yields a $(1 - \epsilon')^3$ approximation to the optimal solution, in time that is polynomial for fixed k and ϵ'. Pick ϵ' so that $(1 - \epsilon')^3 \geq (1 - \epsilon)$ and $\epsilon' \leq \frac{1}{k}$ (we need this for technical reasons). In fact we can set $\epsilon' = \min(\frac{1}{k}, 1-(1-\epsilon)^{\frac{1}{3}})$. Use Apx-Scheme with parameters ϵ' and k, both of which are constant for fixed ϵ. This gives a polynomial time approximation scheme.

2 Sliding Window Algorithm

For completeness we describe the algorithm [8] that applies to the case of identical disks with unit size items.

We keep the data items in a sorted list in non-decreasing order of the number of clients requiring that data item, denoted by R. The list, $R[1], \dots, R[m]$, $1 \leq m \leq M$, 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 clients for a particular data item (i.e., we do not explicitly indicate the current step j of the algorithm in this notation). We assign data items and remove from R the items whose clients are packed completely, and we move the partially packed clients to their updated places according to the remaining number of unassigned clients for that data item.

The assignment of data items to disk d_j has the general rule that we want to select the first consecutive sequence of k or less data items, $R[u], \dots, R[v]$, whose total number of clients is at least the load capacity L. We then assign items $R[u], \dots, R[v]$ to d_j. In order to not exceed the load capacity, we will break the clients corresponding to the last data item into two groups (this will be referred to as *splitting* an item). One group will be assigned to d_j and the other group is re-inserted into the list R. It could happen that no such sequence of items is available, i.e., all data items have relatively few clients. In this case, we greedily select the data items with the largest number of clients to fill d_j. The selection procedure is as follows: we first examine $R[1]$, which is the data item with the smallest number of clients. If these clients 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 examine the total demand of $R[1]$ and $R[2]$, and so on until either we find a sequence of items with a sufficiently large number of clients ($\geq L$), or the first k items have a total number of clients $< L$. In the latter case, we go on to examine the next k data items $R[2], \dots, R[k+1]$ and so on, until either we find k items with a total number of items at least L or we are at the end of the list, in which case we simply select the last sequence of k items which have the greatest total number of clients.

The proof of the tight bound in [4] involves obtaining an upper bound on the number of data items that were not packed in any disk, and upper-bounding the number of clients for each such data item. By using this approach we cannot obtain a *tight* bound for the case when the data items may have differing sizes. One problem with such an algorithm is that it may pack several size 1 items together, leaving out size 2 items for later, and when K is odd, we may waste space on a disk simply because we are left with only size 2 items and cannot pack them perfectly.

3 Multi-List Sliding Window Algorithm

Let M_1 be the number of size-1 items and M_2 be the number of size-2 items. At any stage, let m_1' and m_2' be the number of size-1 and size-2 items on the remaining items list (the list of items whose clients have not been assigned completely). Here we only discuss the case when k is odd, since there is a simple reduction of the case when k is even to the unit size case (as will be shown later).

The algorithm constructs and maintains three lists L_1, L_2 and *aux-list*. If $M_1 < N$, then note that there are at least $N - M_1$ units of unused space in the input instance. In which case, the algorithm adds $N - M_1$ dummy size-1 items with zero load. The algorithm then sorts the size-1 items and the size-2 items in non-decreasing order of demand in lists L_1 and L_2 respectively. The top N size-1 items with the *highest demand* are moved into *aux-list*. The remaining size-1 items are kept in L_1. All the size-2 items are placed in the L_2 list. From this stage on, the algorithm maintains the L_1, L_2 and *aux-list* lists in non-decreasing order of demand.

For each disk (stage), the algorithm must make a selection of items from L_1, L_2 and *aux-list*. Assume the lists are numbered starting from 1. Exactly one item for the selection is always chosen from *aux-list* (see Fig. 1). The algorithm then selects w_1 consecutive items from L_1 and w_2 consecutive items from L_2 such that the total utilized space of the selected items from L_1 and L_2 is $\leq k-1$ ($< k-1$ if we have an insufficient number of items, or the items have a very high density).

Define the *wasted* space of a selection to be the sum of the unused space and the size of the item that must be split to make the selection load-feasible. At each stage the algorithm makes a list of selections ($\mathcal{S}$) by combining the following selections (one from L_2, one from L_1 and one from *aux-list*). It selects w_2, $0 \leq w_2 \leq \min(\lfloor \frac{k}{2} \rfloor, m_2^{'})$ consecutive size-2 items from L_2 at each of the positions $1 \ldots (m_2^{'} - w_2 + 1)$. It selects w_1, $0 \leq w_1 \leq \min(k - 2w_2 - 1, m_1^{'})$ size-1 items from L_1 at each of the positions $1 \ldots (m_1^{'} - w_1 + 1)$. It selects a size-1 item from *aux-list* at each of the positions $1 \ldots |\textit{aux-list}|$.

If $\forall s \in \mathcal{S}, load(s) < L$ the algorithm outputs the selection with highest load. If $\exists s \in \mathcal{S}$ where $load(s) \geq L$, then let $\mathcal{D}$ be the set of all the selections in $\mathcal{S}$ with load $\geq L$. Let $\mathcal{D}^{'} \subseteq \mathcal{D}$ be the set of all the selections which can be made load-feasible by allowing the split of either the highest size-2 item in the selection, or the highest size-1 item from L_1 in the selection, or the size-1 item from *aux-list* in the selection.

The algorithm chooses the $d \in \mathcal{D}^{'}$ with minimum wasted space. The algorithm outputs $d^{'} = \{d_1, \ldots, d_i^{'}\}$ where $d_i = d_i^{'} + d_i^{''}$, $load(d_1, \ldots, d_i) \geq L$ and $load(d_1, \ldots, d_i^{'}) = L$. In the step above, the algorithm is said to split d_i. If $d_i^{''} > 0$ the algorithm then reinserts $d_i^{''}$ (the broken off piece) into the appropriate position in the list from which d_i was chosen. If the broken off piece was reinserted into *aux-list*, the algorithm shrinks the length of *aux-list* by one. The size-1 item that leaves *aux-list* in the previous step is then reinserted into the appropriate position of the L_1 list. If the broken off piece was reinserted into some other list (other than *aux-list*) then note that the size of *aux-list* reduces by one anyway since the item from *aux-list* is used up completely.

4 Analysis of the Algorithm

For each disk in the system, the solution consists of an assignment of data items along with an assignment of the demand (i.e., the clients for this item that are assigned to the disk) for each of the items assigned to the disk. We will argue that the ratio of packed demand (S) to total demand is at least $(1 - \frac{1}{(1+\sqrt{\lfloor k/2 \rfloor})^2})$. Further, we will show that this bound is tight (see Appendix A). This bound is trivial to obtain for even k as shown next. Most of this section will focus on the case when k is odd.

4.1 Even K. Given an instance I create a new instance $I^{'}$ by merging pairs of size-1 items to form size-2 items. If M_1 (the number of size-1 items in I) is odd, then we create a size-2 item with the extra (dummy) size-1 item. Size-2 items in I remain size-2 items in $I^{'}$. Note that since k is even, $I^{'}$ will remain feasible although M_1 may be odd. We now scale the sizes of the items in $I^{'}$ by $1/2$ and apply the sliding window algorithm described in Section 2. The basic idea is to view a capacity k disk as a capacity $k/2$ since each item has size 2. From the result of [4], we get the desired bound of $\frac{S}{U+S} \geq (1 - \frac{1}{(1+\sqrt{\lfloor k/2 \rfloor})^2})$.

It is easy to use the above approach to obtain a bound of $(1 - \frac{1}{k})(1 - \frac{1}{(1+\sqrt{\lfloor k/2 \rfloor})^2})$ when k is odd. However, this bound is not tight.

4.2 Odd K. The algorithm produces a set of load saturated disks at first, where the total load is exactly L. The number of such disks will be referred to as N_l. The number of disks with load less than L will be N_s (non load saturated disks). We will assume that the minimum load on a non load saturated disk is cL (in other words define c appropriately, so that each non load saturated disk has load at least cL). We will refer to $us(i)$ as the *utilized space* on disk d_i. This is the total amount of occupied space on a disk.

We will first bound the space wasted in packing the load-saturated disks and then bound the space wasted in packing the non load-saturated disks to show that $\frac{S}{S+U} \geq (1 - \frac{1}{(1+\sqrt{\lfloor k/2 \rfloor})^2})$.

The algorithm works in stages, producing one window per stage which corresponds to the assignment for a single disk. We know that, at any stage, if we have at least one load saturated window, then the algorithm selects the window with load $\geq L$ that is:

- Load-feasible with one split (i.e. the load of the window becomes $= L$ by splitting at most one item) and

- Minimizes wasted space

L_1 is the list of $(M_1 - N)$ size-1 items, L_2 is the list of size-2 items, and *aux-list* is the list of N size-1 items with highest load.

If at any stage, both the L_1 and L_2 lists are empty while there are some items remaining in the aux-list, since the number of items in the aux-list is equal to the number of unpacked disks, they will be packed completely (this actually follows from [8], see [4] for a simpler proof). Furthermore it is not hard to show that if at any stage j, we have produced $j-1$ load-saturated disks and the total size of the objects in the L_1 and L_2 lists is $\leq k-1$, then all the items will be packed at the termination of the algorithm.

LEMMA 4.1. *When the current window has* $us(i) = k-1$ *and a size 2 item is split, then every leftmost window in the future of size* $k-2$ *(not including the split piece) has load* $\geq L$*,*

This argues that the split piece of size 2 along with a chosen window of size $k-2$ will produce a load saturated disk. If again we split off a piece of size 2, then repeatedly we will continue to output load saturated windows, until we run out of items.

Proof. Assume not. Now w (the current window of size $k-1$) has i items $m_1^1, \ldots, m_1^i$ from L_1, $m_2^1, \ldots, m_2^j$ from L_2 ($j = 0$ implies w has no items from L_2) and *aux-item(1)* from *aux-list* (this item is mandatory). Consider a window (call it $w^{'}$) with size $k-2$ and with load $< L$ chosen in the future. (We will discuss the case when the window is chosen at the next step, however since the items are sorted in non-decreasing order the same proof works for all such windows.) Suppose $w^{'}$ has items say $m_1^{i+1}, \ldots, m_1^{i+i^{'}}$ from L_1 ($i^{'} = 0$ implies $w^{'}$ has no items from L_1), $m_2^{j+1}, \ldots, m_2^{j+j^{'}}$ from L_2 ($j^{'} = 0$ implies $w^{'}$ has no items from L_2) and *aux-item(2)* from *aux-list* (this item is mandatory). Let ℓ_q^p be the number of clients for item m_q^p.

Note the following:

$$\sum_{p=1}^{i} \ell_1^p + \sum_{p=1}^{j} \ell_2^p + \textit{aux-list(1)} \geq L$$

$$\sum_{p=1}^{i} \ell_1^p + \sum_{p=1}^{j-1} \ell_2^p + \textit{aux-list(1)} < L$$

Since we cannot reduce to load to L by splitting a size 1 item, we have

$$\sum_{p=1}^{i-1} \ell_1^p + \sum_{p=1}^{j} \ell_2^p + \textit{aux-list(1)} > L$$

Suppose the window of size $k-2$ we select has load $< L$. This implies that

$$\sum_{p=i+1}^{i+i'} \ell_1^p + \sum_{p=j+1}^{j+j'} \ell_2^p + \textit{aux-list(q)} < L$$

Since the items of a list are in non-decreasing order, we can claim the following:

$$\sum_{p=1}^{i'} \ell_1^p + \sum_{p=1}^{j'} \ell_2^p + \textit{aux-list(1)} < L$$

Call this window w''. It has size $k-2$ and load $< L$. There are three cases based on the values of j and j'.

1. $j = j'$**.** Since $j = j'$ and $i' = i-1$, we obtain

   $$\sum_{p=1}^{i-1} \ell_1^p + \sum_{p=1}^{j} \ell_2^p + \textit{aux-list(1)} < L$$

   This is in direct contradiction to the assumption we made about w (see equation above).

2. $j > j'$**.** Add $m_2^{j'+1}$ to w''. This window now has size k. If the load now is $> L$, we can find a window of load $> L$ with size k that is load saturating. This is a contradiction to our choice of a window of size $k-1$. Otherwise, we keep adding items from L_2 and dropping items from L_1, to maintain a size k window, until we obtain a window with load $> L$.

   (Certainly by the time we add m_2^j we obtain a window of size k with total load $> L$.) As soon as this happens we have found a window with size k that is load saturating. This is a direct contradiction to our choice of a window of size $k-1$.

3. $j < j'$**.** Add $m_1^{i'+1}$ and $m_1^{i'+2}$ to w''. If the load now is $> L$ then we can load saturate with a window of size $\geq k-1$ and split a size 1 item. This is in contradiction to the choice that we made. Now assume that the total load is $\leq L$ and the size is exactly k. We remove $m_2^{j'}$ from w'' and add $m_1^{i''+3}$ and $m_1^{i''+4}$. Again if the load $> L$ we are done. We keep doing this until the load exceeds L. This must happen after we remove m_2^{j+1}.

The proof of the following lemma is similar to the proof for Lemma 4.1 and we omit it due to space constraints.

LEMMA 4.2. *When the current window has* $us(i) \leq k-2$ *and an item is split, then every leftmost window of the same size as the current window must have load* $\geq L$

We next show that for each load saturated disk we have at most two units of wasted space.

LEMMA 4.3. *If at the termination of the algorithm there are unassigned clients then for every load-saturated disk d_i one of the following conditions must hold:*

1. *Disk d_i has $us(i) \geq k-1$ and a size-1 item is split, or*
2. *Disk d_i has $us(i) = k$ and a size-2 item is split.*

Proof. We need to show that if we produce a load-saturated disk that violates conditions (1) and (2) then all the items from all the lists (L_1, L_2 and *aux-list*) will be packed completely.

From Lemma 4.1, we know that if we waste three units of space by splitting an item of size 2 and having $us(i) = k-1$ then we will assign all clients to disks.

From Lemma 4.2 we know that when the current window has ≥ 2 units of unused space and a size-1 item is split or a size-2 item is split, then every leftmost window of the same size as the current window must have load $\geq L$.

Since we know that every leftmost window with the same size as the current window has load $\geq L$, we also know that in the next stage there exists a window of the same size as the current window with load $\geq L$. Further, since the current window has size $\leq k-2$, the broken off piece from the current window can be reused in the next stage. As a result, we will produce load-saturated disks until the total load of the items remaining on L_1 and L_2 is $< L$. However the total size of the items remaining on L_1 and L_2 is now $< size(current-window) \leq k-2$. In this case, as mentioned previously, all the items will be packed in the following rounds.

LEMMA 4.4. *If at the termination of the algorithm there are unassigned clients then either*

1. *All the non load-saturated disks are size-saturated.*
2. *Only size-2 items are remaining and there is at most one non load-saturated disk with exactly one unit of unused space and all the other non load-saturated disks are size-saturated, or*

Proof. If at the termination of the algorithm, L_1 is not empty then all the non-load saturated disks *must* also be size-saturated; otherwise the algorithm would have found a selection with higher load by adding in another item from L_1.

Now consider the case where L_1 is empty and L_2 is not empty. Since L_2 is not empty, for each non-load saturated disk i we have $us(i) \geq k-1$. Now assume (for contradiction) that there are two non-load saturated disks i and j (say $i < j$) s.t. $us(i) = us(j) = k-1$. If we have $us(i) = k-1$ then L_1 must become empty after this selection has been assigned to i; otherwise, the algorithm could just have added in another item from L_1 and would have found a selection with higher load. Since $us(i) = k-1$, the L_1 list becomes empty after the current selection has been assigned to disk i. Now L_1 is empty and exactly one item from *aux-list* will be forced onto j, so for all future disks $j > i$, $us(j)$ must be odd. Since k is odd and we have $us(j) \geq k-1$, it follows that $us(j) = k$ and we have a contradiction.

THEOREM 4.1. *It is always possible to pack a $(1 - \frac{1}{(1+\sqrt{\lfloor \frac{k}{2} \rfloor})^2})$-fraction of items for any instance.*

Proof. As a result of the Lemmas 4.3 and 4.4, we know that at the termination of the algorithm if there are unassigned clients then either:

1. At most $2N_l + 1$ units of space are wasted in the packing and only size-2 items are remaining, or
2. At most $2N_l$ units of space are wasted in the packing.

We will show that in both cases the total load of the remaining items (U) is $\leq \frac{N_l c L}{\lfloor \frac{k}{2} \rfloor}$.

We first see how to prove the claim using this bound. The number of satisfied clients (S) is at least $L \times N_l + c \times N_s \times L$. Subtracting this quantity from the upper bound on the load of the input instance $(N \times L)$ gives us $U \leq (1-c) \times N_s \times L$ where U is the unassigned clients. Hence the ratio of unpacked (U) to packed (S) items can be bounded as follows.

$$\frac{U}{S} \leq \frac{\min(\frac{N_l \times c \times L}{\lfloor \frac{k}{2} \rfloor}, (1-c) \times N_s \times L)}{L \times N_l + c \times N_s \times L}$$

Since

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

the claimed bound now follows from the method outlined in Appendix C to upper bound $\frac{U}{S}$.

1. If at most $2N_l + 1$ units of space are wasted in the packing and only size-2 items are remaining, then we can have at most N_l size-2 items on the remaining items list. Let the load on the lightest loaded non load-saturated disk be cL. Since any non load-saturated disk must have at least $\lfloor \frac{k}{2} \rfloor$ size-2 groups (i.e. either two size-1 items or a single size-2 item), the load on the lowest size-2 group is at most $\frac{cL}{\lfloor \frac{k}{2} \rfloor}$ (average load of an assigned item).

The load of any size-2 item on the remaining items list must be $\le \frac{cL}{\lfloor \frac{k}{2} \rfloor}$ since otherwise, the algorithm could have obtained a better packing by swapping the size-2 item on the remaining items list with this lowest size-2 group. Therefore, the total load of the remaining items is $\le \frac{N_l cL}{\lfloor \frac{k}{2} \rfloor}$.

2. Let $m_1^{'}$ be the number of size-1 items on the remaining items list, and let $m_2^{'}$ be the number of size-2 items on the remaining items list. We know that all the non load-saturated disks have k units of utilized space. This disk has $\lfloor \frac{k}{2} \rfloor$ size-2 groups (i.e. either two size-1 items or a single size-2 item) and a size-1 item. Let the load on this size-1 item be x.

   - If $m_1^{'} = 0$, then the same reasoning as for case 1 gives us the desired bound.
   - If $m_1^{'} = 1$. Since we know that all the non load-saturated disks are size-saturated, we have at least $\lfloor \frac{k}{2} \rfloor + 1$ objects (both size-1 and size-2 items) on the lightest loaded disk. Therefore, the maximum load of the smallest object on the lightest loaded disk is $\le \frac{cL}{\lfloor \frac{k}{2} \rfloor + 1}$. The load of the single size-1 item on the remaining items list must be at most x and must also be $\le \frac{cL}{\lfloor \frac{k}{2} \rfloor + 1} \le \frac{cL}{\lfloor \frac{k}{2} \rfloor}$ since otherwise, the algorithm would have obtained a better packing by swapping the size-1 item on the remaining items list with the lowest object (a size-1 or size-2 item) on the lightest loaded disk.

$$
\begin{aligned}
U &\le \min(x, \frac{cL}{\lfloor \frac{k}{2} \rfloor}) + m_2^{'}(\frac{cL - x}{\lfloor \frac{k}{2} \rfloor}) \\
&\le \min(x, \frac{cL}{\lfloor \frac{k}{2} \rfloor}) + (\lfloor \frac{2N_l - 1}{2} \rfloor)(\frac{cL - x}{\lfloor \frac{k}{2} \rfloor}) \\
&\le \min(x, \frac{cL}{\lfloor \frac{k}{2} \rfloor}) + (N_l - 1)(\frac{cL - x}{\lfloor \frac{k}{2} \rfloor}) \\
&\le \frac{N_l cL}{\lfloor \frac{k}{2} \rfloor}
\end{aligned}
$$

   - If $m_1^{'} \ge 2$. Let L_1^i be the remaining load of the i^{th} size-1 item and let L_2^j be the remaining load of the j^{th} size-2 item. Since $m_1^{'} \ge 2$, we must have that load of any unpacked size-2 group be less than the load of the smallest size-2 group on the lightest loaded disk. We can thus obtain a bound for $\sum_{i=1}^{m_1'} L_1^i$ as follows. Consider all pairs of size 1 items with load $L_1^i + L_1^j$ with $i \ne j$. The total load for this pair cannot exceed $\frac{cL-x}{\lfloor \frac{k}{2} \rfloor}$, which is the load for the minimum size 2 group that was packed. Summing over all pairs gives

$$\sum_{(i,j) i \ne j} (L_1^i + L_1^j) = (m_1' - 1) \sum_{i=1}^{m_1'} L_1^i.$$

Thus

$$(m_1' - 1) \sum_{i=1}^{m_1'} L_1^i \le \frac{m_1'(m_1' - 1)}{2} \frac{cL - x}{\lfloor \frac{k}{2} \rfloor}.$$

Simplifying yields

$$\sum_{i=1}^{m_1'} L_1^i \le \frac{m_1'}{2} \frac{cL - x}{\lfloor \frac{k}{2} \rfloor}.$$

$$
\begin{aligned}
U &\le \sum_{i=1}^{m_1'} L_1^i + \sum_{j=1}^{m_2'} L_2^j \\
&\le m_1^{'}. \min(x, \frac{1}{2}(\frac{cL - x}{\lfloor \frac{k}{2} \rfloor})) + m_2^{'}(\frac{cL - x}{\lfloor \frac{k}{2} \rfloor}) \\
&\le \frac{(m_1^{'} + 2m_2^{'})}{2} . \frac{cL}{\lfloor \frac{k}{2} \rfloor} \\
&\le \frac{N_l cL}{\lfloor \frac{k}{2} \rfloor}
\end{aligned}
$$

5 Generalized Sliding Window Algorithm

The sizes of the items in our instance are chosen from the set $\{1, \ldots, \Delta\}$. In this section, we present an algorithm that guarantees to pack a $\frac{k-\Delta}{k+\Delta}\left(1 - \frac{1}{\left(1+\sqrt{\frac{k}{2\Delta}}\right)^2}\right)$ fraction of clients for any valid problem instance.

The algorithm works in two phases. In the first phase it produces a solution for a set of N disks each with storage capacity $k + \Delta$ and load capacity L. In the second phase, the algorithm makes the solution feasible by dropping items from these disks until the subset of items on each disk has size at most k and load at most L, respectively.

In the first phase of the algorithm, the algorithm keeps the items in a list sorted in non-decreasing order of density ρ_i, where $\rho_i = \frac{l_i}{s_i}$, l_i and s_i are the load and size of item i. At any stage of the algorithm, this list will be referred to as the list of remaining items.

For each disk, the algorithm *attempts* to find the first (from left to right in the sorted list) "minimal" consecutive set of items from the remaining items list such that the load of this set is at least L and the total size of the items in the set is at most $k+\Delta$. We call such a consecutive set of items a "minimal" load-saturating set. The set is "minimal" because removing the item with highest density (i.e., the rightmost item) from this set will cause the load of the set to become less than L. Say the items in such a "minimal" set are some $x_u, \ldots, x_v$. We have $\sum_{i=u}^{v} l_i \geq L$, $\sum_{i=u}^{v-1} l_i < L$, $\sum_{i=u}^{v} s_i \leq k+\Delta$ and u is the first index where such a load-saturating set can be found. If a "minimal" load-saturating set is found, then the algorithm breaks the highest density item in this set (i.e., x_v) into two pieces $x_{v'}$ and $x_{v''}$ such that $l_{v'} + \sum_{i=u}^{v-1} l_i = L$. The piece $x_{v''}$ is reinserted into the appropriate position on the remaining items list.

If the algorithm is unable to find such a "minimal" load-saturating set, then it outputs the last (from left to right) "maximal" consecutive set of the highest density items from the remaining items list. We call such a set a "maximal" non load-saturating set. Say the items in this "maximal" set are some $x_p, \ldots, x_q$ (where x_q is the last item on the list of remaining items at this stage). The set is "maximal" in the sense that $s_{p-1} + \sum_{i=p}^{q} s_i > k+\Delta$ (if x_p is not the first item in the list of remaining items) and $\sum_{i=p}^{q} s_i \leq k+\Delta$. Since we know that the set was not a load-saturating set we have $\sum_{i=p}^{q} l_i < L$.

The algorithm outputs these sets as follows. Let the items on the remaining items list be $x_1, \ldots, x_q$. For each disk, add item x_1 to the current selection. Repeat the following steps until we find either a "minimal" load-saturating set or a "maximal" non load-saturating set: Say the next item, that is the item on the remaining items list after the last item in current selection, at any stage is x_i. If $load$(current selection) $< L$ and $s_i + size$(current selection) $\leq k+\Delta$, then add x_i to current selection. Else if $load$(current selection) $< L$ and $s_i + size$(current selection) $> k+\Delta$, drop the lowest density items from current selection as long as $s_i + size$(current selection) $> k+\Delta$, and then add x_i to current selection. Note that if $load$(current selection) $\geq L$ or $x_i = \emptyset$, then we have found either a "minimal" load-saturating set or a "maximal" non load-saturating set. If the algorithm finds a "minimal" load-saturating set then it breaks off the highest density item in current selection (as described above), reinserts the broken-off piece into the appropriate position on the remaining items list and outputs the modified current selection. If the algorithm finds just a "maximal" non load-saturating set, it simply outputs the current selection. After the algorithm outputs a selection, these items are removed from the list of remaining items. At the end of the first phase of the algorithm, each disk is assigned either a "minimal" load-saturating set of items or a "maximal" non load-saturating set of items.

In the second phase, for each disk, the algorithm drops the lowest density items assigned to the disk until the size of the packing is at most k. Since the load of the packing was feasible to begin with, at the end of this phase the algorithm produces a feasible solution.

THEOREM 5.1. *It is always possible to pack a* $\frac{k-\Delta}{k+\Delta}\left(1 - \frac{1}{\left(1+\sqrt{\frac{k}{2\Delta}}\right)^2}\right)$ *fraction of clients for any valid input instance.*

LEMMA 5.1. *If $us(i) \leq k-\Delta$ for any load-saturated disk i at the end of phase I of the algorithm, then all items are packed at the end of phase I of the algorithm.*

Proof. Suppose that the items assigned to a load-saturated disk i are $x_1, \ldots, x_p$ (in non-decreasing order of density). Then we have $\sum_{j=1}^{p} s_j \leq k-\Delta$, $\sum_{j=1}^{p} l_j \geq L$ and $\sum_{j=1}^{p-1} l_j < L$.

If the items in our current selection are not the first p items, then clearly some items were dropped. Since some items were dropped, adding in another item to the current selection must have made $size$(current selection) $> k+\Delta$. The algorithm will then drop items to make the current selection size feasible. Since each item has size at most Δ, this operation cannot decrease $us(i)$ below k. We have a contradiction since we assumed $us(i) \leq k-\Delta$. So the only way for the selection to have size $\leq k-\Delta$ is for the selection to consist of some p items where these p items are also the first p items in the list of remaining items.

As a result, every consecutive subset of items of size between $k-\Delta$ and k has load $> L$. Since the algorithm permits every disk to pack items of total size upto $k+\Delta$ in phase I, note that the broken off piece (which has size $\leq \Delta$) from the previous load-saturated disk can be accomodated in the next load-saturated disk. In this way, the algorithm produces load-saturated disks until there are no more items in the remaining items list.

LEMMA 5.2. *At the end of phase I of the algorithm, at least a* $1 - \frac{1}{\left(1+\sqrt{\frac{k}{2\Delta}}\right)^2}$ *fraction of clients are always packed.*

Proof. We will argue that the total unassigned load at the end of phase I is less than $\frac{2\Delta N_l cL}{k}$ where N_l is the number of load-saturated disks in the assignment and cL denotes the load of the lightest loaded non load-saturated disk in the assignment. The bound will then follow from the method outlined in Appendix C. (Note that the bound we use there is $\frac{N_l cL}{\lfloor \frac{k}{2} \rfloor}$.) Observe that at the end of phase I of the algorithm, every item that has unassigned load (i.e., every item on the list of remaining items at the end of phase I of the algorithm) will have density less than that of the lowest density item on lightest loaded disk. This is because when we are unable to produce any more load-saturated disks, the algorithm effectively outputs the largest possible consecutive set of the highest density items. Let items $x_1, \ldots, x_p$ be assigned to the lightest loaded non load-saturated disk. Since the load of the lightest loaded disk is cL, we also have $\sum_{i=1}^{p} load(x_i) = cL$. Since in phase I we allow each disk to be filled upto size $k + \Delta$, we have $k < \sum_{i=1}^{p} size(x_i) \leq k + \Delta$, unless all the items have been packed. Let ρ_{min} denote the density of the lowest density item assigned to the lightest loaded disk. Then we have:

$$\begin{aligned}\rho_{min} \sum_{i=1}^{p} size(x_i) &\leq \sum_{i=1}^{p} \rho_i \cdot size(x_i) = cL \\ \rho_{min} &\leq \frac{cL}{\sum_{i=1}^{p} size(x_i)} \\ \rho_{min} &< \frac{cL}{k}\end{aligned}$$

Now for each item y_i on the remaining items list, since the density of y_i is less than ρ_{min}, we have $load(y_i) \leq \rho_{min} \cdot size(y_i)$. So the total load of the items on the remaining items list is $\leq \rho_{min} \cdot \sum size(y_i)$. Since at the end of phase I the remaining items list was not empty, from Lemma 5.1, we know that each load-saturated disk is filled to size $> k - \Delta$. Further, we know that each non load-saturated disk is filled to size $> k$, otherwise we can add an item to this disk. Since our instance was feasible, the total size of all the items in the instance is Nk. Every time we create a load-saturated disk we might split at most one item and this item can have size at most Δ. As a result, the size of the unpacked items is:

$$\sum size(y_i) \leq Nk + N_l\Delta - N_l\,(k - \Delta) - N_s k = 2N_l\Delta$$

where each y_i is an item on the remaining items list, N_l is the number of load-saturated disks and N_s is the number of non load-saturated disks. So the total unassigned load (i.e. the total load of the items on the remaining items list at the end of phase I of the algorithm) is $\leq \rho_{min} \cdot 2\Delta N_l < \frac{2\Delta N_l cL}{k}$.

Let S' be the total load of items packed at the end of phase II and let S be the total load of items packed at the end of phase I.

Lemma 5.3. *At the end of phase II of the algorithm,* $\frac{S'}{S} \geq \frac{k-\Delta}{k+\Delta}$.

Proof. Say the items assigned to a disk are $x_1, \ldots, x_p$ (these items are labeled in non-decreasing order of density). Suppose $\sum_{j=1}^{p} size(x_j) > k$. Say items $x_1, \ldots, x_q$ need to be dropped from the selection to make $\sum_{j=q+1}^{p} size(x_j) \leq k$. Since the largest sized item in our instance has size Δ, $\sum_{j=1}^{q} size(x_j) \leq 2\Delta$ and $\sum_{i=q+1}^{p} s_i > k - \Delta$. Let ρ be the density of item x_{q+1}. Since the items $x_1, \ldots, x_p$ are labeled in non-decreasing order of density, for each disk we can lower bound the remaining load (after dropping items $x_1, \ldots, x_q$ to make it size-feasible) as follows:

$$\begin{aligned}\sum_{i=q+1}^{p} \rho_i s_i &\geq \rho \sum_{i=q+1}^{p} s_i \\ &> \rho\,(k - \Delta)\end{aligned}$$

Further, for each disk we can upper bound the lost load as follows:

$$\sum_{i=1}^{q} \rho_i s_i \leq \rho\,(2\Delta)\,.$$

Therefore, the ratio of total lost load to total remaining load is at most $\frac{2\Delta}{k-\Delta}$ and the fraction of total load remaining after phase II is at least $\frac{k-\Delta}{k+\Delta}$ (using Equation 3.1 in Appendix C).

Using these two lemmas, we easily obtain the proof of Theorem 5.1.

6 Polynomial Time Approximation Schemes

Note that

$$f(k, \Delta) > \frac{k - \Delta}{k + \Delta}\left(1 - \frac{1}{\frac{k}{2\Delta}}\right) > \frac{k - \Delta}{k + \Delta}\left(1 - \frac{1}{\frac{k-\Delta}{2\Delta}}\right).$$

Thus algorithm SW-Alg2 can definitely pack a $\frac{k-\Delta}{k+\Delta}\left(1 - \frac{1}{\frac{k-\Delta}{2\Delta}}\right)$ fraction of items for any valid problem instance. Also note that $\frac{k-\Delta}{k+\Delta}\left(1 - \frac{1}{\frac{k-\Delta}{2\Delta}}\right)$ tends to 1 as $k \to \infty$.

If $1 - \epsilon \leq \frac{k-\Delta}{k+\Delta}\left(1 - \frac{1}{\frac{k-\Delta}{2\Delta}}\right)$ then we can use Algorithm SW-Alg2 and get a solution within the desired error bounds. If $1 - \epsilon > \frac{k-\Delta}{k+\Delta}\left(1 - \frac{1}{\frac{k-\Delta}{2\Delta}}\right)$ then k is a constant ($k < \frac{2(2-\epsilon)\Delta}{\epsilon}$) and we develop a PTAS for this case. This scheme is a generalization of the scheme developed in [4]. Algorithm Apx-Scheme takes as input parameters k, c and ϵ' and produces a solution that has an approximation factor of $(1-\epsilon')^3$, in time that is polynomial for fixed $\epsilon' > 0$ and integers k, c. The sizes of the items are in the set $\{a_1, \ldots, a_c\}$ with $a_i \geq 1$. (If the sizes are chosen from $\{1, \ldots, \Delta\}$ for some constant Δ, then this is easily seen to be the case.) To get a $(1-\epsilon)$ approximation, we simply define $\epsilon' = 1 - (1-\epsilon)^{\frac{1}{3}}$.

For technical reasons we will also need to assume that $\epsilon' \le \frac{1}{k}$. If this is not the case, we simply lower the value of ϵ' to $\frac{1}{k}$. Since k is a fixed constant, lowering the value of ϵ' only yields a better solution, and the running time is still polynomial.

The approximation scheme involves the following basic steps:

1. Any given input instance can be approximated by another instance I' such that no data item in I' has an extremely high demand.
2. For any input instance there exists a near-optimal solution that satisfies certain structural properties concerning how clients are assigned to 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.

6.1 Preprocessing the Input Instance. We say that an instance I is *B-bounded* if the size of each set U_j is at most B. We omit the proof of the following lemma as it is almost the same as in [4].

LEMMA 6.1. *For any instance I, we can construct in polynomial time another instance I' such that*

- *I' is (αL)-bounded,*
- *any solution S' to I' can be mapped to a solution S to I of identical value, and*
- $|\mathsf{OPT}(I')| \ge (1-\epsilon')|\mathsf{OPT}(I)|$.

6.2 Structured Approximate Solutions. Let us call a data item j *unpopular* if $|U_j| \le \epsilon' \frac{L}{k}$, and *popular* otherwise. For a given solution, we say that a disk is *light* if it contains less than $\epsilon' L$ clients, 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 popular data items and between heavy disks and unpopular data items, obeys some nice properties. The proof of the following lemmas is in [4].

LEMMA 6.2. *For any instance I, there exists a solution S satisfying the following properties:*

- *at most one light disk receives clients from a set U_j.*
- *a heavy disk is assigned either zero or all clients that require an unpopular item.*
- *S packs at least $(1-\epsilon')\mathsf{OPT}(I)$ items.*

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

LEMMA 6.3. *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 popular data item, and*
- *S' packs at least $(1-\epsilon')|S|$ items.*
- *each heavy disk packs $(1-\epsilon')L$ items corresponding to popular items.*

6.3 The Approximation Scheme. We start by preprocessing the given input instance I so as to create an (αL)-bounded instance I' as described in Lemma 6.1. We now give an algorithm to find a solution S to I' such that S satisfies the properties described in Lemmas 6.2 and 6.3 and packs the largest number of clients. Clearly,

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

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 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 disks in $\mathcal{H}$ and $\mathcal{L}$ respectively.

Packing items in $\mathcal{H}$: We first guess a vector $\langle l_1, l_2, ..., l_{N'}\rangle$ such that $l_i = \langle l_i^1, \ldots, l_i^c\rangle$ where l_i^j denotes the number of unpopular size a_j data items whose clients are assigned (completely) to a disk $d_i \in \mathcal{H}$.

Since all disks are identical, we can guess each such vector in $O(N^{k+1^c})$ time by guessing a compact representation of the following form. First note that the number of possible distinct l_i vectors is upper-bounded by $(k+1)^c$, simply because each l_i^j value is chosen from the set $\{0, 1, \ldots, k\}$. (Note that better bounds can be derived since to be a feasible packing we require that $\sum_j l_i^j a_j \le k$.) Let $T^{(1)}, T^{(2)}, \ldots, T^{(\gamma)}$ be distinct feasible vectors. We guess a vector $\langle q_0, q_1, \cdots, q_\gamma\rangle$ such that $\sum_{i=0}^{\gamma} q_i = N'$ where q_i denotes the number of disks in $\mathcal{H}$ that are of type $T^{(i)}$. 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 disk d_i the largest size l_i^j size a_j unpopular data items that remain.

Next we develop a dynamic program moving across the disks from 1 through N' so as to find an optimal (ϵ'^2/k)-integral solution for packing the largest number of clients from the popular data items.

For the purpose of this packing, the capacity of each heavy disk is restricted to be $(1-\epsilon')L$ and the number of data items allowed in disk d_i is given by $k - \sum_j l_i^j a_j$, since we already packed l_i^j unpopular items of size a_j in d_i..

Let $\beta = k/\epsilon'^3$ and $q = \lceil(\epsilon'^2 L)/k\rceil$. The dynamic program is based on maintaining a β-tuple $\vec{v} = \langle v_1^1, v_2^1, ..., v_\beta^1, v_1^2, v_2^2 \ldots, v_\beta^2, \ldots, v_1^c, v_2^c, \ldots, v_\beta^c \rangle$ where v_i^j denotes the number of size a_j popular data items that have $i \cdot q$ clients 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 clients 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 items, the total number of state vectors is bounded by $(Nk)^{ck/\epsilon'^3}$, which is polynomial for any fixed ϵ'.

Packing items in $\mathcal{L}$: We know that our solution need not assign clients corresponding to a popular data item to more than one disk in $\mathcal{L}$. Moreover, at most $\epsilon' L$ clients from any popular data item are packed in a disk in $\mathcal{L}$. So at this stage we can truncate down the size of each popular data item to $\lfloor \epsilon' L \rfloor$. Together with the unpopular items, we have c lists of items, L_i' $(i = 1 \ldots c)$where L_i' has both popular and unpopular items of size a_i. (0, the popular items are truncated as mentioned above.)

We have exactly $N - N'$ disks that are light disks, and we wish to obtain an optimal packing of these light disks using the c lists mentioned above. First note that if $\epsilon' \leq \frac{1}{k}$ then no subset of data items of total size at most k can ever load saturate a disk. This essentially implies that we can ignore the load dimension, only worrying about the storage capacity constraint. However, at the same time we wish to pack a set of data items that yield the maximum number of assigned clients.

Our approach is based on the following idea. For each $i = 1 \ldots c$ we guess n_i, the number of data items from L_i' that are chosen to be packed in light disks. Since there are $O(M^c)$ such choices, this is a polynomially bounded search space. For each such 1, we can easily compute the "yield" of this guess, namely the number of clients that can be assigned if we can pack n_i data items from each list L_i' in the $N - N'$ light disks. Note that within each list L_i' we will always choose the most profitable set of n_i items (with the maximum number of clients).

We still need an algorithm to verify if it is possible to pack n_i items from each list L_i'. This is done as follows. We can characterize each disk by a vector $(x_1, x_2, \ldots x_c)$ where x_i is the number of items of size a_i packed in this disk. For this to be feasible, it must satisfy the property that $\sum_{i=1}^c a_i x_i \leq k$. Note that this immediately upper bounds the value of x_i by $\lfloor \frac{k}{a_i} \rfloor$. The number of possible vectors is thus at most $O(k^c)$, in other words a constant for fixed k and c. Hence we obtain the fact that each light disk is characterized by a constant number of (feasible) types $T^{(1)}, \ldots, T^{(\alpha)}$ where $T^{(j)} = (x_1^j, \ldots, x_c^j)$.

Let N_i be the number of disks of type $T^{(i)}$. Clearly, we are looking for a solution to the following Integer Program (IP):

$$\sum_{j=1}^{\alpha} N_j = N - N'$$

$$\sum_{j=1}^{\alpha} x_i^j N_j = n_i \forall i = 1 \ldots c$$

The first constraint simply specifies that the total number of disks of each type is exactly the total number of light disks. The second constraint says that exactly n_i items of each size a_i are packed. Since this is an integer program with a constant number of variables, we can use the algorithm by Lenstra [5] to solve it, or we can use the fact that each N_i is upper bounded by $N - N'$ to obtain a polynomial time algorithm.

Acknowledgements: We thank Moses Charikar, Leana Golubchik and An Zhu for useful discussions.

References

[1] S. Berson, S. Ghandeharizadeh, R. R. Muntz, and X. Ju. Staggered Striping in Multimedia Information Systems. *SIGMOD*, pages 79–90, 1994.

[2] C. Chekuri and S. Khanna. On multidimensional packing problems. In *ACM Symp. on Discrete Algorithms*, pages 185–194, 1999.

[3] M. Dawande, J. Kalagnanam, and J. Sethuraman. Variable Sized Bin Packing With Color Constraints. Technical report, IBM Research Division, T.J. Watson Research Center, 1999.

[4] L. Golubchik, S. Khanna, S. Khuller, R. Thurimella, and A. Zhu. Approximation algorithms for data placement on parallel disks. In *ACM Symp. on Discrete Algorithms*, pages 223–232, 2000.

[5] H. W. Lenstra. Integer programming with a fixed number of variables. *Math. of Oper. Res.*, pages 538–548, 1983.

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

[7] H. Shachnai and T. Tamir. Polynomial time approximation schemes for class-constrained packing problems. Proceedings of Workshop on Approximation Algorithms (APPROX), pages 238–249, 2000.

[8] H. Shachnai and T. Tamir. On two class-constrained versions of the multiple knapsack problem. *Algorithmica*, Vol. 29, Issue 3, pages 442–467, 2000.

[9] M. Stonebraker. A Case for Shared Nothing. *Database Engineering*, 9(1):4–9, 1986.

[10] J. Wolf, H. Shachnai, and P. Yu. DASD Dancing: A Disk Load Balancing Optimization Scheme for Video-on-Demand Computer Systems. In *ACM SIGMETRICS/Performance Conf.*, pages 157–166, 1995.

A Motivational Application.

Recent advances in high speed networking and compression technologies have made multimedia services, such as video-on-demand (VOD) servers, feasible. 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 [1]. However, this approach suffers from a number of implementation-related shortcomings.

An alternate system is described in [10] where the nodes are connected in a shared-nothing manner [9]. 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. It is these issues that give rise to our data placement problem.

B Tight Example

The tight example for $k = 2p+1$ and $k = 2p$ is the same and the same as in [4]. In both cases the input instance will only consist of size-2 items. Note that with such an instance for $k = 2p + 1$, N units of space can never be used if the input instance was feasible.

We now give an example to show that the bound of $(1 - \frac{1}{(1+\sqrt{\lfloor k/2 \rfloor})^2})$ is tight. In other words, there are instances for which no solution will pack more than $(1 - \frac{1}{(1+\sqrt{\lfloor k/2 \rfloor})^2})$ fraction of items. Assume that $\lfloor \frac{k}{2} \rfloor$ is a perfect square, where k is the storage capacity of a disk. Let N the number of disks be $1 + \sqrt{\lfloor \frac{k}{2} \rfloor}$ and let $L = \lfloor \frac{k}{2} \rfloor + \sqrt{\lfloor \frac{k}{2} \rfloor}$. There are $\lfloor \frac{k}{2} \rfloor$ size-2 items with a large demand (call them "large items"). Say these items are $U_1, \ldots, U_{\sqrt{\lfloor \frac{k}{2} \rfloor}}$ each with demand $2 + \sqrt{\lfloor \frac{k}{2} \rfloor}$. There are also $(\lfloor \frac{k}{2} \rfloor - 1)(1 + \sqrt{\lfloor \frac{k}{2} \rfloor}) + 1$ size-2 items with a small demand (call them "small items"). Say these items are $U_{\sqrt{\lfloor \frac{k}{2} \rfloor}+1}, \ldots, U_{\lfloor \frac{k}{2} \rfloor (1+\sqrt{\lfloor \frac{k}{2} \rfloor})}$.

We will show that at least $\sqrt{\lfloor \frac{k}{2} \rfloor}$ demand will never get packed. In this case, the fraction of unpacked items is at least $\frac{\sqrt{\lfloor \frac{k}{2} \rfloor}}{(1+\sqrt{\lfloor \frac{k}{2} \rfloor})(\lfloor \frac{k}{2} \rfloor + \sqrt{\lfloor \frac{k}{2} \rfloor})}$ which is exactly

$\frac{1}{(1+\sqrt{\lfloor \frac{k}{2} \rfloor})^2}$. This proves the claim.

First consider the $\sqrt{\lfloor \frac{k}{2} \rfloor}$ large items. An unsplit item U_i has all its demand allocated to a single disk. A split item U_i has its demand allocated to several disks. For a disk that contains at least one large unsplit item, the available load capacity is at most $\lfloor \frac{k}{2} \rfloor - 2$. Note that after packing one large unsplit item, the available load capacity is smaller than the storage capacity. Even is there is no single large unsplit item on a disk, we can obtain the same configuration without losing any packed demand by swapping the demand of this item with the demand of the other items on the disk. The disks now have one large unsplit item and at most $\lfloor \frac{k}{2} \rfloor - 2$ small items. The remaining disks have only large split items. Assume that there are exactly $p (0 \leq p \leq \lfloor \frac{k}{2} \rfloor)$ large items that do not get split $U_1, \ldots, U_p$ with disk d_i containing U_i.

Consider the remaining $N-p$ disks; we are left with at least $\lfloor \frac{k}{2} \rfloor \times N - p(\lfloor \frac{k}{2} \rfloor - 1) = \lfloor \frac{k}{2} \rfloor \times (N-p) + p$ items, but we only have $\lfloor \frac{k}{2} \rfloor \times (N-p)$ storage capacity left. Since the remaining $\lfloor \frac{k}{2} \rfloor - p$ large items are all split, this generates an additional $\lfloor \frac{k}{2} \rfloor - p$ instances of items. Thus we have at least $\lfloor \frac{k}{2} \rfloor \times (N-p) + p + \lfloor \frac{k}{2} \rfloor - p$ items. This will create an excess of $\lfloor \frac{k}{2} \rfloor$ items that we cannot pack.

C Proof Details

Proof. The ratio of unpacked (U) to packed (S) items is at most

$$\frac{U}{S} \leq \frac{\min(\frac{N_l \times c \times L}{\lfloor \frac{k}{2} \rfloor}, (1-c) \times N_s \times L)}{L \times N_l + c \times N_s \times L}$$

Let $y = \frac{N_l}{N}$ and thus $1 - y = \frac{N_s}{N}$. Simplifying the upper bound above we obtain.

$$\frac{U}{S} \leq \frac{\min(\frac{cy}{\lfloor \frac{k}{2} \rfloor}, (1-c)(1-y))}{y + c(1-y)}$$

$$\frac{U}{S} \leq \min(\frac{\frac{cy}{\lfloor \frac{k}{2} \rfloor}}{y + c(1-y)}, \frac{(1-c)(1-y)}{y + c(1-y)})$$

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, we need to set the two terms equal, which means

$$\frac{cy}{\lfloor \frac{k}{2} \rfloor} = (1-c)(1-y)$$

$$y = \frac{1-c}{1-c+\frac{c}{\lfloor \frac{k}{2} \rfloor}}$$

Substituting for y gives us that the upper bound for this ratio is at most $\frac{c-c^2}{\lfloor \frac{k}{2} \rfloor - \lfloor \frac{k}{2} \rfloor c + c^2}$. This achieves its maxima

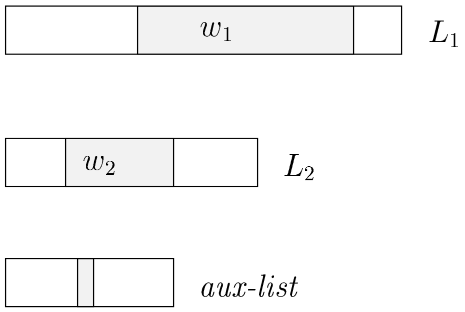

Figure 1: Figure to show lists used by Algorithm.

when $c = (1 - \frac{1}{1+\sqrt{\lfloor \frac{k}{2} \rfloor}})$. The fraction of all the items that are packed is

$$\frac{S}{U+S} = \frac{1}{1+\frac{U}{S}} \tag{3.1}$$

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