ABSTRACT

Title of dissertation: ALGORITHMS FOR DATA PLACEMENT, RECONFIGURATION AND MONITORING IN STORAGE NETWORKS

Srinivas Raaghav Kashyap
Doctor of Philosophy, 2007

Dissertation directed by: Professor Samir Khuller
Department of Computer Science

In this thesis we address three problems related to self-management of storage networks - data placement, data reconfiguration and data monitoring. Examples of such storage networks include centrally managed systems like Storage Area Networks and Network Attached Storage devices, or even highly distributed systems like a P2P network or a Sensor Network.

One of the crucial functions of a storage system is that of deciding the placement of data within the system. This data placement is dependent on the demand pattern for the data and subject to constraints of the storage system. For instance, if a particular data item is very popular the storage system might want to host it on a disk with high bandwidth or make multiple copies of the item. We present new results for some of these data placement problems.

As the demand pattern changes over time, the storage system will have to modify its placement accordingly. Such a modification in placement will typically involve movement of data items from one set of disks to another or changing the number of copies of a data item in the system. For such a modification to be effective, it should be computed and applied quickly since the system is running inefficiently during this reconfiguration. We propose new schemes to reconfigure the data placement to deal with changing demand.

To re-compute data placement periodically and to reconfigure the data placement, we need to continuously track of the demand distribution in the storage system and also be able to answer aggregate queries about the demand distribution. The data monitoring portion of the thesis deals with such problems that arise in the context of distributed data management applications. A monitoring system for such a scenario would need to process large amounts of data from a widely distributed set of data sources. The thesis presents new schemes that improve communication-efficiency of existing methods that address these problems.

# ALGORITHMS FOR DATA PLACEMENT, RECONFIGURATION AND MONITORING IN STORAGE NETWORKS

by

Srinivas Raaghav Kashyap

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2007

Dedicated

To *amma* and *appa*.

Acknowledgements

First and foremost, I thank my advisor and mentor Samir Khuller. He introduced me to the field of algorithms and theoretical computer science. A single guest lecture he gave during an undergraduate algorithms class I was enrolled in completely changed my view of algorithms. He inspired me to join graduate school and has taught me everything I know about research and algorithms. His amazing insight into problems which is second to none and his ability to simplify ideas have always helped me understand better. His approach to research has been a source of great inspiration for me. He is one of the most patient people I have ever known and was always available to discuss even the most trivial matters. He has much more to offer than one could expect from an academic advisor and was someone I could turn to for any kind of assistance or guidance. I would like to thank Samir for supporting me through NSF grant CCF-0430650 and NSF award CCR-0113192.

I thank Rajeev Rastogi for hosting me for over a year during the course of two internships at Bell Labs. I benefited greatly from his extensive experience and excellent intuition about the right problems to study. A significant portion of the results in this thesis are a consequence of my interactions with Rajeev and the other great folks I worked with at Bell Labs Bangalore - Anand Srinivasan, Supratim Deb, K. V. M. Naidu and S. R. Jeyashankher.

I thank Rajiv Gandhi for support and guidance during the early years of graduate school. I thank Atif Memon and David Doermann for their support. Apart from the folks above, I would like to thank Srinivasan Parthasarathy, Indrajit Bhattacharya, Krishna Chitrapura, Yung-Chun (Justin) Wan, Leana Golubchik, Vijay Gopalakrishnan, Vinayaka Pandit and Julian Mestre. Technical discussions and interactions with them were a lot of fun and have benefited me greatly. The results in the thesis have also benefited from discussions with and comments from Suman Banerjee, David Mount, Moses Charikar and An Zhu. Much thanks to Aravind Srinivasan, Amol Deshpande, Peter Keleher and Mark Shayman for agreeing to be on my advisory committee.

I thank my roommates and friends during my stay in Maryland - Nitin Chandrachoodan, Nagarajan Valanoor, Gaurav Aggarwal, Abheek Anand, Akhil Gupta, Narayanan Ramanathan, Shiv N. P. Vitaladevuni and Saurabh Srivastava. I thank my friend from school, Bharath Madhusudan, for encouraging me to come to College Park in the first place! I thank Pratima Rao, an inspiring teacher who taught me Computer Science in school.

Above all, I thank my family - Amma (Nalini Iyengar) and Appa (A. V. Narasimha ”Nachu” Iyengar) who have constantly given me everything they have, always put my needs before theirs and encouraged me to pursue my goals; Kavitha Babu (my wife) for being patient, keeping me motivated, helping me through difficulties and for being there all the time; Sheela Jacob and Babu Jacob for their patience and support. I thank my cousin and friend, Jayant Sirohi, who took care of me when I was new to Maryland and even well after that - I also thank him for Starcraft, for Monty Python and whole lot of other things! I thank my friends and relatives who have helped me in so many ways at various points in my academic life.

Table of Contents

# Chapter 1

# Introduction

The "How much information?" study produced by the school of information management and systems at the University of California at Berkeley [127], estimates that about 5 exabytes of new information was produced in 2002. It estimates that the amount of stored information doubled in the period between 1999 and 2002. It is believed that more data will be created in the next five years than in the history of the world. Clearly we live in an era of data explosion. This data explosion necessitates the use of large storage networks, where a collection of storage elements are made available to a large network of users.

Storage Area Networks (or SANs) and Network Attached Storage (NAS) devices are common examples of enterprise storage systems. Storage area networks (SANs) may be centralized, where the storage area network can contain many heterogeneous storage elements (or disk drives) connected to a single storage space. The storage space can be treated as a black box so that administration of storage is easy. SANs may also be distributed, where the storage area network contains many geographically dispersed disk drive networks. All the constituent networks are treated as one unit and are connected by the iSCSI storage area network protocol over a common storage fabric. In a NAS device the storage elements are typically at a common location. For our purposes both SAN and NAS devices would be examples of a centralized storage network because they are both *centrally managed*.

While in many scenarios centralized storage networks are a natural solution since they simplify management and help in meeting requirements such as consistency and persistence, there are other scenarios where they are not the best solution. For example, the data many have only short-term state or the data itself may be naturally distributed. In such cases, decentralized storage networks are an attractive solution since they avoid single-points of failures, scale naturally, and can leverage resources from participating users, thus avoiding the need for an expensive centralized system. Such fully decentralized storage networks borrow ideas from overlay networking, distributed computing and distributed data management. An example of a decentralized storage network would be a P2P network or a sensor network.

Note that "centralized" and "decentralized" are used to distinguish between how the storage network is managed. While the problems that arise in these two classes of storage networks are varied and diverse, "self-management" is a common goal shared by most of these problems. Self-management means that the system automatically optimizes itself in response to changes in the load, system status, and environment. *This thesis studies three crucial problems that arise in the context of self-management of both centralized and decentralized storage networks.* Two fundamental algorithmic challenges in a storage network are that of data placement and reconfiguration - deciding the placement of data and assignment of requests for data in the storage network; and dealing with changes in the access pattern for the data as well as changes to the storage network itself. To address either of these problems, the problem of monitoring access statistics (for instance answering aggregate queries over access statistics and continuous tracking of access statistics) must be addressed.

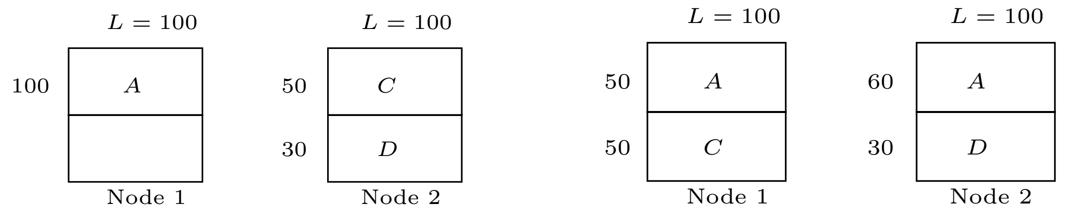
To make the relationship between these problems clearer, consider the following examples.

**Example 1.1:** Consider a sensor network comprising of a large number of sensor nodes. One of the models for querying historical data in this sensor network (beyond "dumb querying") is to view the network as a database that supports archival query processing where queries are pushed inside the network. Such a model is feasible since current trends challenge the conventional wisdom about the role of storage in sensor networks (See [61, 130]). They make a compelling case for equipping sensor nodes with high-capacity energy-efficient local flash storage and redesigning algorithms to exploit cheap storage for reducing expensive communication. Making effective use of data in such a sensor network will require scalable, self-organizing, and energy-efficient data dissemination mechanisms. Promising methods that address these issues include data-centric storage mechanisms (GHTs [160], DHTs [9]).

Consider decentralized storage networks that use such data-centric storage mechanisms as a means of organizing data inside the network. The decentralized data placement portion of this thesis extends such data-centric storage mechanisms to support sophisticated information retrieval applications in storage networks. The decentralized data reconfiguration portion of this thesis presents a low-overhead adaptive replication scheme to ensure that the load incurred while processing queries for data is shared equitably across all the nodes in the storage network. The data monitoring portion of the thesis addresses questions related to answering aggregate queries in decentralized storage networks. For instance, "What is the total number of accesses for this data item across all sensor nodes?" and "Issue an alert whenever the storage space used on any sensor node exceeds a threshold". Answering such queries in a communication efficient manner is clearly essential in addressing both the data placement and reconfiguration problems.

**Example 1.2:** Consider a Video-On-Demand system. 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 via a high-speed network. An alternative system is described in [172] where the nodes are connected in a shared-nothing manner [165]. 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.

Requests are made to a central location that then assigns connections to nodes based on the location of the movies on the nodes and available bandwidth on the nodes. The data placement problems seeks to find such an assignment of movies to nodes and an assignment of connections to nodes (see Figure 1.1). The data reconfiguration results address the problem of rearranging movies and reassigning connections when the demand for movies changes with time. The data



**(a)** A sub-optimal placement that satisfies a total demand of 180.

**(b)** An optimal placement that satisfies a total demand of 190.

**Figure 1.1:** An instance of the data placement problem consisting of two nodes. Each node has storage capacity of 2 and bandwidth capacity of 100. There are also 4 data items. Each item has unit size. The demand for item **A=110**, item **B=10**, item **C=50**, item **D=30**. The figures show two possible data placements.

reconfiguration results can also be used to address the problem of recovering from a scenario where some nodes in the storage network have failed and there is a need to quickly rearrange the data placement to deal with the failures without having to recompute or make wholesale changes to the data placement. Keeping track of the demand distribution in this case is straightforward since all requests are processed through a central location. However, the system might need to enforce certain policies - such as allowing at most a certain number of connections from a certain group of IPs; or monitoring Quality of Service requirements by ensuring that packet loss along network paths to preferred users does not exceed a certain threshold. The data monitoring results can be used to address these problems.

## 1.1 Organization and Contributions

Briefly, this thesis makes the following contributions to the problems of data placement, reconfiguration and monitoring. The precise problem definitions along with detailed results and discussion are presented in subsequent chapters. Previous work is discussed in sections 1.2, 1.3, 1.4, 1.5 and 1.6 of this chapter.

1. Results from the centralized data placement portion of this thesis (Chapter 2) include a polynomial time approximation scheme (PTAS) for the data placement problem - computing

an assignment of data items to nodes and a corresponding assignment of requests to data items while maximizing the profit of assigned requests subject to the capacity constraints of the nodes. The results also include practical combinatorial algorithms for this problem with provably good performance guarantees that hold regardless of the input distribution. These results generalize the results of Golubchik *et al.* [70] to handle the case where data items may have arbitrary sizes.

2. The centralized data reconfiguration portion of the thesis (Chapter 3) considers the problem of adapting an existing data placement to accommodate changes in demand pattern. The thesis introduces a new approach to this problem, by trying to make changes to the existing data placement so that the resulting placement will be the best possible placement that can be obtained within a specified number of migration rounds. In each migration round, nodes are paired up and paired nodes can exchange data during that round. One of our results is that the problem is NP-Hard and we present heuristics for the problem. The thesis demonstrates, through a set of extensive experiments, that even in a small number of consecutive rounds the existing placement for the old demand pattern can be transformed into one that is almost as good as the best placement for the new demand pattern.

3. The decentralized data placement and reconfiguration portion of this thesis (Chapter 4) makes the following contributions:

   (a) Distributed hash table (DHT) based storage mechanisms are popular solutions to the problem of decentralized data placement but lack support for similarity search. Results from this chapter present new algorithms for finding a data placement in *any* DHT based decentralized storage network so that semantically related data can be located easily and is placed on "nearby" nodes. Results include analytical guarantees for the performance of the algorithms in terms of search accuracy and cost. Results from simulations confirm the insights derived from these analytical models.

   (b) The chapter also presents adaptive replication and randomized lookup schemes for *any*

DHT based storage mechanism. These schemes ensure that the number of copies of a data item is proportional to its demand and that all replicas are equally likely to serve a given request. Therefore, for DHT based storage mechanisms, these schemes address the problem of modifying an existing data placement to accommodate changes in demand pattern.

One-shot queries and continuous queries are important classes of queries that arise in data monitoring applications. Communication efficient algorithms for answering these queries in a decentralized setting are essential components in algorithms for data placement and reconfiguration (see Examples 1.1 and 1.2).

4. The one-shot querying portion of the thesis (Chapter 5) presents a novel gossip-based scheme that improves the result of Kempe *et al.* [107] using which all the nodes in an $n$ node overlay network can compute the common aggregates of MIN, MAX, SUM, AVERAGE, and RANK of their values using $O(n \log\log n)$ messages within $O(\log n \log\log n)$ rounds of communication. This is the first result that shows how to compute these aggregates with high probability using only $O(n \log\log n)$ messages.

5. In the continuous querying portion of this thesis (Chapter 6), we introduce a new set of methods called non-zero slack schemes for communication efficient monitoring of distributed SUM queries and also undertake a comprehensive study of these non-zero slack schemes.

   (a) We show both analytically and empirically that non-zero slack schemes outperform the state-of-the-art zero slack scheme for different data distributions.

   (b) We present adaptive algorithms for setting threshold values at remote nodes in the presence of non-zero slack (for changing data distributions).

   (c) Finally, we present the results of a thorough and detailed set of experiments using both synthetically generated data and real world data, and show that our adaptive non-zero slack algorithms can result in significant savings in the amount of communication.

This is the first work to systematically study non-zero slack schemes for detecting distributed constraint violations.

The following sections will put the contributions of this thesis in perspective with previous work for each of these problems. Recall that we use the terms centralized and decentralized to distinguish between how the storage network is managed. Section 1.2 discusses the centralized data placement problem, Section 1.3 discusses the centralized data reconfiguration problem, Section 1.4 discusses the decentralized data placement problem, Section 1.5 discusses the decentralized data reconfiguration problem, Section 1.6.1 discusses the problem of answering one-shot queries and Section 1.6.2 discusses the problem of answering continuous queries.

## 1.2 Data Placement in Centralized Storage Networks

Most work in this area assumes that a data item can fit on a node completely in terms of storage size. Our thesis makes this assumption as well. This is a reasonable assumption given that storage devices have much larger capacities than the size of individual data items. Also note that this assumption does not rule out the use of striped storage on a lower level - for example a node in our data placement problem can be a “logical disk” that comprises of several physical disks in say a RAID configuration where the data items assigned to the logical disks (nodes) are striped across the physical disks for performance and reliability.

In the centralized scenario, constituent nodes of the storage network might be heterogeneous but they are typically co-located. Therefore the connection cost to serve a request for a data item is similar across all the nodes and depends only on where the request originated from within the network. However in some cases, like with distributed SANs, the connection cost to serve a request for a data item depends both on the node that stores the data item and also on where the request originated from within the network. The former is the version of the problem *without* connection costs (Problem 1.1) and the latter is the version of the problem *with* connection costs (Problem 1.2). This thesis makes contributions to the version of the problem *without* connection costs. As observed earlier, most SAN and NAS configurations fall under this category and the version *without* connection costs is applicable in most centralized data placement scenarios. This version is also a special case of the version *with* connection costs. Consequently, the best possible results for this problem are better than the best possible results for the version *with* connection costs.

### 1.2.1 Centralized data placement *without* connection costs

This thesis, in context of the centralized data placement problem, makes contributions to the variant of the problem that can be abstracted as follows:

**Problem 1.1:** We are given a collection of $M$ data items that need to be assigned to a storage network consisting of $N$ nodes $d_1, \ldots, d_N$. Data item $i$ has size $s_i$. Each node $d_j$ is characterized 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 requests that it can serve. We are also given a set of $U$ requests. Each request $u \in U$ seeks a particular data item $i$ and has profit $f_u$ associated with the data item that it seeks. The goal is to find a placement of data items to nodes and an assignment of clients to nodes to maximize the total profit of requests served, subject to the capacity constraints of the storage network.

This variant is identical to the "connection cost" variant (Problem 1.2) except for that in this case, the profit associated with a request is uniform across all the nodes whereas it could vary with the node that it was assigned to in the "connection cost" variant.

Shachnai and Tamir [156] 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. They gave an elegant algorithm, called the *sliding window* algorithm, and showed that this algorithm packs all items whenever $\sum_{j=1}^{N} C_j \geq M + N - 1$. They showed that the problem is NP-hard when each node has an arbitrary load capacity, and unit storage. Golubchik *et al.*[70] improved this result to show that even the problem with identical nodes is NP-hard for any fixed $k \geq 2$ ($C_j = k$ for all nodes $j$). Golubchik *et al.*[70] also establish a tight upper and lower bound on the number of items that can *always* be packed for any input instance regardless of the distribution of requests for data items under the assumption that all items require unit storage. They also present a PTAS for the problem when all items require unit storage.

Packing problems with color constraints are studied in [47, 155]. 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 there is a constraint on the number 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.

In this thesis (Chapter 2), we generalize the result of Golubchik *et al.* [70] to the case where data items may have arbitrary sizes. Specifically, for the case where $s_i \in \{1, \ldots, \Delta\}$ for some constant $\Delta$, 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 $k$. In addition we develop an algorithm for which we can prove tight upper and show a matching lower bound when $s_i \in \{1, 2, 2^2, \ldots, \Delta\}$ and $\log \Delta \in Z$ regardless of the input distribution. Independently, Shachnai and Tamir [157] have recently announced a result similar to ours. However, the algorithms and the ideas in their work are based on a very different approach as compared to the ones taken in this thesis. The results presented in this portion of the thesis have appeared in [105].

For the sake of completeness, previous work on the version of the problem with connection costs is also presented here.

### 1.2.2 Centralized data placement *with* connection costs

The problem can be abstracted as follows :

**Problem 1.2:** We are given a collection of $M$ data items that need to be assigned to a storage network consisting of $N$ nodes $d_1, \ldots, d_N$. Data item $i$ has size $s_i$. Each node $d_j$ is characterized 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 total bandwidth of requests that may be assigned to it. We are also given a set of $U$ requests. Each request $u \in U$ seeks a particular data item, requires bandwidth $b_u$ and has profit $f_{uj}$ associated with each node $j$ for the data item that it seeks. The goal is to find a placement of data items to nodes and an assignment of requests to nodes to maximize the total profit of requests served, subject to the capacity constraints of the storage network.

Baev and Rajaraman [17] study the problem of data placement in arbitrary networks. They formalize a minimization version in which they need to place objects in caches to minimize the total connection costs. They give a constant-factor approximation for this problem, which is improved to factor 10 by Swamy in [166]. However in their formulation, nodes *do not* have bandwidth constraints. Meyerson *et al.*[122] study the version *with* bandwidth as well as storage constraints and present a $O(\log n)$ approximation algorithm that requires node capacities to be enlarged by a $O(1)$ factor. Guha *et al.*[76] present a solution for this problem that is a constant factor approximation but requires node capacities to be enlarged by a $O(\log n)$ factor.

Korupulu *et al.*[112] study the problem of data placement in hierarchical networks (a version of the problem where distances have properties similar to an ultrametric). Their problem again ignores bandwidth constraints and they show that the problem can be solved using min-cost flow under the hierarchical network distances assumption. They also present a faster local search based 2-approximation (minimization).

Fleischer *et al.*[58] call this the *Distributed Caching Problem.* Their results show LP-based $(1 - 1/e - \epsilon)$ approximation algorithm and a local search algorithm with $(1/2 - \epsilon)$ approximation guarantee. They also present complementary lower bounds showing that this problem cannot be approximated better than $(1 - 1/e)$ unless $NP \subseteq DTIME\left(n^{o(\log\log n)}\right)$, even if there exists an exact poly-time algorithm for the single-bin subproblem. However, the algorithms they present are not very practical, both the local search and the separation oracle based LP solution that they devise use a polynomial-time approximation scheme for the single-node subproblem as a subroutine. None of the polynomial time approximation schemes for the problem known in the literature are very practical. Hence there is a space for practical combinatorial algorithms with a good approximation ratio.

## 1.3 Data Reconfiguration in Centralized Storage Networks

Maintaining an optimal or near optimal data placement is crucial in ensuring that the storage network operates efficiently. The optimal data placement is likely to change over time because of changes in access pattern for the data items, addition of nodes or failures of nodes. Consequently, the storage network will have to modify its data placement to adapt to these changes. Such a modification will typically involve movement of data items from one set of nodes to another or requires changing the number of copies of a data item in the system. For such a modification to be effective, it should be computed and applied quickly since the system runs at sub-optimal efficiency during this phase. The data reconfiguration portion of the thesis deals with the problem of finding such an efficient modification.

Seo *et al.*[154] study the disk replacement problem (DRP) which is concerned with finding a sequence of disk drive removals and additions to obtain a final, target storage system while minimizing the data migration cost. Results from their work can be used to address the data migration issues that arise out of changes to the underlying storage network. However, their work is not concerned with changes to the access pattern of the data items. Their work treats all data items as being similar. They also do not have explicit bandwidth or storage constraints on the disks. They consider a system in which the data items are load balanced in both the initial and final configurations.

Hewlett-Packard's AutoRAID [171] consists of a small disk array that supports a two-level RAID hierarchy. The system uses adaptive replication to handle changes in the storage system due to disk additions and also to handle changes in access pattern for the data items. For replacing disks, the system first removes the old disks, then attaches all the new disks and waits for the automatic data reorganization and rebalancing to complete (without interrupting its operations).

Golubchik *et al.*[71], Hall *et al.*[82] and Khuller *et al.*[109] study the data migration problem. They use a data placement algorithm to compute a new target layout. The goal of the data migration problem is to convert the existing layout to the target layout as quickly as possible. The communication model they assume is a half-duplex model where a matching on the nodes can be fixed, and for each matched pair one can transfer a single object in a round. The goal is to minimize the number of rounds taken. Khuller *et al.*[109] develop constant factor approximation algorithms for this NP-hard problem. In practice these algorithms find solutions that are reasonably close to optimal. However, even when there is no drastic change in the demand distribution it can still take many rounds of migration to achieve the new target layout. This happens since the scheme completely disregards the existing placement in trying to compute the target placement.

In the centralized data reconfiguration portion of this thesis we consider a new approach to the problem of dealing with changes in the demand pattern. We consider the following problem. Given a certain number of migration rounds (a node may be involved in at most one transfer per round), we want to obtain a layout by making changes to the existing layout so that the resulting layout will be the best possible layout that can be obtained within the specified number of rounds. Of course, such a layout is interesting only if it is significantly better than the existing layout for the new demand pattern.

The approach used in this thesis to address the problem of finding a good layout that can be obtained in a specified number of rounds by finding a sequence of layouts. Each layout in the sequence can be transformed to the next layout in the sequence by applying a small set of changes to the current layout. These changes are computed so that they can be applied within one round of migration. Results in this thesis show that by making these changes even for a small number of consecutive rounds, the existing placement for the old demand pattern can be transformed into one that is almost as good as the best layout for the new demand pattern. The method can therefore be used to quickly transform an existing placement to deal with changes in the demand pattern. Results from this portion of the thesis appeared in [106].

## 1.4 Data Placement in Decentralized Storage Networks

The goal of this body of work is to find a decentralized solution to Problem 1.2. While there has been some progress toward this goal, the solutions are not very practical. In this subsection, we will briefly review some previous results for this problem including results which indicate that the problem is hard to solve in a decentralized way. The decentralized solutions presented in previous work are known to be PLS-complete (See [95, 137, 150] for more about PLS-completeness). Consequently, there have been attempts at solving a much simpler version of the original problem. The contributions of this thesis are related to solutions for the simplified problem (Problem 1.3).

### 1.4.1 Decentralized solutions for Problem 1.2

Goemans *et al.*[67] formulate Problem 1.2 (ignoring bandwidth constraints) as a market sharing game. In this game, in each step, every node (one at a time) greedily modifies the subset of items stored on it to maximize its individual profit. If no subset of items has higher profit than the profit of items already stored on the node, the node makes no changes. Among the nodes that contain a particular data item, each request for the data item is assigned to the node with the lowest connection cost for that request. The profit that accrues to a node is the sum of the individual profits associated with the requests that are served by the subset of items stored on the node. The total profit of a solution is the sum of the individual profits of all the nodes. A locally optimal solution is one where *none* of the nodes have an incentive to change the subset of items that they have decided to store. They show that such a locally optimal solution always exists.

For the case where all data items have the same size, finding the subset of items that maximizes profit is an easy problem. In this case, the decentralized procedure converges to a locally optimum solution in a polynomial number of steps. They show that the total profit of a locally optimal solution always has at least $1/2$ of the profit of the globally optimal solution. However, this is not a very practical solution since the polynomial time algorithm they present requires each node to know about the total number of requests for *each* of the data items in order to make a decision about which subset of items to store on it.

They also extend their results to the case where data items can have arbitrary sizes. However, now any node can only find a subset of items that *approximates* the subset of items with maximum profit. They show that in this case their decentralized algorithm reaches a locally optimal solution that always has at least $1/\log n$ of the profit of the globally optimal solution. While they show that the algorithm achieves this solution in finitely many steps, they do not show that it does so in a polynomial number of steps. Again, this is not a very practical solution.

Fleischer *et al.*[58] extends the formulation of this game to handle bandwidth constraints. As before, in each step, every node (one at a time) greedily modifies the subset of items stored on it to maximize its individual profit. If no subset of items has higher profit than the profit of items already stored on the node, the node makes no changes. As before, a locally optimal solution is one where none of the nodes have an incentive to change the subset of items that they store. They show that for some instances, such a locally optimal solution might not exist at all. They also show that for some instances, even though a locally optimal solution exists, the nodes may not be able to find this local optimum in a polynomial number of steps. Clearly, the problem appears to be very difficult.

### 1.4.2 A simpler problem

To simplify the problem and obtain practical schemes that might do well, many attempts at solving this problem drop the explicit "objective function" that we are trying to maximize here. Another simplification is to drop the hard capacity constraints, instead the goal is to balance resource usage across all the nodes. These assumptions make the problem considerably easier. This simpler problem can be abstracted as follows:

**Problem 1.3:** We are given a set of $N$ nodes $Q = \{d_1, \dots, d_N\}$ that are completely interconnected. Let $U = \{1, \dots, p\}$ represent the set of all numbers available for addressing data items. All data items are assumed to be of equal size. Only a subset $D \subseteq Q$ of nodes and only a subset $M \subseteq U$ of the data items may be present at a time. Each data item $m \in M$ has to be assigned to a unique node $j \in D$. Each node $j$ has a limited storage capacity $C_j$, representing the number of data items that can be stored on it. The goal is then to find, using a decentralized scheme, a data placement such that the data items in $M$ are distributed among the nodes $D$ so that the maximum number of data items stored on any node is minimized (and less than $C_j$ for each node $j$).

There is a vast amount of literature related to Problem 1.3 in the peer-to-peer community. A complete discussion of results from this area are beyond the scope of this thesis. Distributed Hash Tables or DHTs are a common solution to the problem and they provide an efficient distributed data placement and lookup mechanism. Common examples of DHT based peer-to-peer systems include but are not limited to CAN[144], CHORD[164], Viceroy[128, 118], Koorde[99], Pastry[149], Tapestry[179], Kademlia[121], Kelips [79], Skipnet[83], Salad[54] and Bamboo[148]. Most of these systems draw on the distributed hashing schemes presented in Karger *et al.* [100], Plaxton *et al.* [141], Brinkmann *et al.* [24, 25], Schindelhauer *et al.* [152], Adler *et al.* [5], Schiedler [151] and Awerbuch *et al.* [152].

However such DHTs only provide basic lookup functionality. Several researchers have proposed mechanisms to extend the scope of DHTs beyond the traditional lookup. Reynolds *et al.*[147], Liu *et al.*[114], and Shi *et al.*[161] address efficient keyword searching in DHTs. The work of Gupta *et al.*[77] and Schmidt *et al.*[153] use SPHs to distribute high dimensional data vectors on top of a CHORD overlay. The former supports approximate range queries while the latter supports exact range queries. The work of Bhattacharjee *et al.*[21] supports efficient set intersection operations using view trees. The pSearch system [167] extends a specific DHT to support similarity searching.

Results from the distributed data placement portion of this thesis extend the capabilities of *any* such distributed hash based lookup mechanism to extend its functionality to support similarity based lookup services. The differences between the pSearch system and ours are discussed in detail in the distributed data placement portion of the thesis. Enabling similarity search on DHTs is an important step in the direction of enabling more realistic and distributed IR applications over a distributed storage network. The results presented in this portion of the thesis have appeared in [22].

## 1.5 Data Reconfiguration in Decentralized Storage Networks

While the DHT based decentralized data placement schemes presented in Section1.4.2 for Problem 1.3 ensure that the maximum storage utilized on any node is minimized, they do not make any attempt to balance load incurred by answering requests for data items. These systems therefore need some form of caching or replication to achieve load balance when the data items have non-uniform popularities. Decentralized data reconfiguration addresses this issue by spawning or reducing the number of replicas of data items based on the popularities of the data items.

With the exception of the hashing scheme of Adler *et al.*[5] which is tied to a specific underlying DHT, most other DHTs provide only for static replication where each object in the DHT is replicated a fixed number of times and hence they do not deal with changing query distributions. The Lightweight Adaptive Replication (LAR) protocol of Gopalakrishnan *et al.*[72] addresses this problem by measuring the load on individual servers and using the load measurements to create appropriate number of copies of a data item. They also modify the DHT lookup primitive by augmenting nodes in the DHT with information about the newly created copies.

The adaptive replication technique presented in this thesis also relies on server load information for spawning and retracting copies of a data item. However, our scheme differs from that of LAR in significant ways. The differences are discussed in detail in the distributed data reconfiguration portion of the thesis. One of the most attractive features of the scheme presented in this thesis is that the scheme can operate over *any* underlying DHT (most adaptive replication schemes are tied to properties of a particular underlying DHT). Results from this portion of the thesis have appeared in [22].

## 1.6 Data Monitoring

The problem of centralized data management - that of organizing, indexing, accessing and querying data that is located centrally is well understood. However, in distributed scenarios, these data management problems assume a different character since it is not feasible to collect the data in one place: the volume of data collection is too high, and the capacity for data communication relatively low. For example, in battery-powered wireless sensor networks, the main drain on battery life is communication, which is orders of magnitude more expensive than computation or sensing. This establishes a fundamental tenet for distributed data monitoring: push computational work into the network to reduce communication.

The data monitoring portion of this thesis deals with such distributed data monitoring problems and efficient solutions for such data monitoring problems are crucial building blocks in algorithms for the data monitoring and reconfiguration problems. There are two broad classes of data monitoring problems in previous work. In the *one shot* model, a query is issued by a user at some node, and must be answered based on the current state of data in the network. In the *continuous* model, a query is placed by a user which requires the answer to be available continuously. The thesis addresses problems from each of these models and presents new schemes that improve communication-efficiency of existing methods that address these problems.

### 1.6.1 Data Monitoring (One-shot Queries)

One-shot queries are interested in computing the aggregate (e.g., MAX, SUM, QUANTILES, RANK of an element) of a collection of data that is distributed across an $n$-node overlay network. Common assumptions are that time is slotted and synchronized across nodes, messages exchanged in any round cannot be arbitrarily large and that in any round each node can only communicate with a $O(1)$-fraction of the nodes.

A popular approach to answering one-shot queries efficiently is to use in-network query processing. Nodes are organized in a tree structure. Each node hears from all children, computes the aggregate and sends the result to its parent (each node sends only one item). Of course, in case of evaluating duplicate sensitive aggregates (for example quantiles), the algorithms need to use sketch summaries or other mergeable summaries. These sketches or summaries are easy to merge and functions evaluated on these sketches are guaranteed to be a good approximation to the exact answer. The choice of which sketch to use depends on the function or aggregate of interest. A complete discussion of such techniques is beyond the scope of this thesis. Commonly used sketch techniques include: CM sketches [40], FM sketches [57], AMS sketches [11, 10] and Bloom filters. Commonly used mergeable summaries include those of Greenwald *et al.*[74] and Manjhi *et al.*[119].

The tree aggregation techniques mentioned above assume a reliable network, however this is not a valid assumption in sensor networks or peer-to-peer networks. Nath *et al.*[129] present a aggregation scheme to compute MAX that is robust to such failures. The idea is to broadcast each node's value to multiple nodes instead of a single node. The procedure relies on the MAX aggregate being order and duplicate insensitive (ODI). Nath *et al.*[129] also extend FM sketches (which are also ODI) to compute sum aggregates in a robust manner. In general, one can take sketches and other summaries and make them ODI by replacing counters in them with FM sketches (which are useful to keep track of the number of distinct items). Examples include the schemes presented in Cormode *et al.*[41], Marios *et al.*[81] and Considine *et al.*[36]. Manjhi *et al.*[119] use multi-path routing of ODI summaries to achieve robust computation of aggregates.

However, these techniques may not be very practical in sensor network scenarios since sketches with FM sketches for counters can grow very large and also due to the computational cost for sensors in executing these schemes. To overcome the above-mentioned problems, several researchers have proposed decentralized gossip-based schemes for computing various aggregates in overlay networks [107, 30, 93, 129, 23]. In gossip-based protocols, each node exchanges information with a randomly chosen communication partner in each round. By their very nature, gossip-based schemes are robust; they are resilient to message failures as well as node failures, thus making them ideally suited for P2P, wireless and sensor networks with potentially poor link-reliability.

Kempe *et al.*[107] presented the first set of analytical results on computation of aggregate functions using randomized gossip. They analyzed a simple gossip-based protocol for computing sums, averages, quantiles and other aggregate functions. In their scheme for estimating averages, each node selects another random node to which it sends half of its value; a node on receiving a set of values just adds them to its own halved value. Kempe *et al.*showed that these values converge to the true average in $O(\log n)$ rounds resulting in $O(n \log n)$ messages.

This thesis, addresses the following question: *is it possible to reduce the message complexity of aggregate (MAX, SUM, AVERAGE, RANK of an element) computation schemes from* $O(n \log n)$ *while relaxing the number of rounds to slightly exceed* $\log n$*?*. The thesis presents a novel scheme to compute MIN, MAX, SUM, AVERAGE and RANK using $O(\log n \log \log n)$ rounds of communication and $O(n \log \log n)$ messages. This is the first result that computes these various aggregates in a network with probabilistic link and node failures using only $O(n \log \log n)$ messages. Thus, compared to previous work [107], our scheme achieves a significant reduction in communication overhead at the cost of only a modest increase in the number of rounds. Results from this portion of the thesis have appeared in [104].

### 1.6.2 Data Monitoring (Continuous Queries)

The goal of this body of work is to continuously track a global query over a distributed set of data streams. Typically there a number of nodes that are each connected to a coordinator. Other communication models are possible as well but this is the standard communication model used in previous work. There is a need to guarantee or compute an answer at the coordinator that is always correct or within some guaranteed accuracy bound. Naïve solutions must continuously centralize all data but this incurs enormous communication overhead. Sometimes periodic polling suffices for simple tasks. However very frequent polling causes high communication and infrequent polling causes delays in observing events. Techniques that reduce communication while guaranteeing rapid response to events are desirable.

Within this framework of continuous, distributed, and resource-constrained systems, there are many possible types of continuous monitoring queries that can be posed. The research community has looked at developing algorithms for computing and tracking a wide range of aggregate statistics over distributed data streams, including top-k [16], quantiles [39], MAX[162], set-expressions[46] and joins[38]. These apply to a general class of continuous monitoring applications, where the goal is to optimize the operational resource usage of these algorithms and still guarantee that the estimate of the aggregate function is always within specified error bounds.

Many queries rely on monitoring sums or counts of values, in combination with thresholds (lower bounds) and is the focus of the continuous query monitoring portion of this thesis. Olston *et al.* [135] study the problem of continuously tracking multiple SUM queries with different error bounds. The problem we consider, of minimizing communication overhead while maintaining accurate counts *above* a threshold, is not covered by this approach. To monitor thresholded sums in a communication efficient manner, the idea of distributed triggers was introduced by Jain *et al.*in [91] and further explored by Sharfman *et al.*[158], Huang *et al.*[87, 86, 85], Agrawal *et al.*[6] and Keralapura *et al.*[108]. They describe distributed constraints monitoring or distributed triggers as a mechanism of reducing the amount of communication. These methods filter out “uninteresting” events and do not require communication across the network for these events; thus, reducing the communication needed to perform the computations. Huang *et al.* [97] consider a novel variant of the instantaneous tracking problem where they track constraint violations that *persist over time*. Please see Chapter 6 for a detailed discussion of the differences between these results and the results in this thesis.

This thesis introduces a new set of methods called non-zero slack schemes to implement distributed SUM queries efficiently. It shows, both analytically and empirically, that these methods can lead to a considerable reduction in the amount of communication. It proposes three adaptive non-zero slack schemes that adapt to changing data distributions; the best scheme is a lightweight reactive scheme that probabilistically adjusts local constraints based on the occurrence of certain events (using only a periodic probability estimation). It conducts an extensive experimental study using real-life and synthetic data sets, and shows that the non-zero slack schemes presented in this thesis incur significantly less communication overhead compared to the state of the art zero slack scheme (over a 60% savings). Results from this portion of the thesis have appeared in [103].

# Chapter 2

# Centralized Data Placement

This is joint work with Samir Khuller. These results also appeared in [105].

## 2.1 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 $N$ nodes $d_1, d_2..., d_N$. We are also given sets $U_1, U_2, ..., U_M$ such that $U_i$ is the set of clients seeking the $i$th data item. Each data item has size $s_i$. Each node $d_j$ 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 nodes and an assignment of clients to nodes 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 nodes. 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 nodes where $C_j = k$ and $L_j = L$ for all nodes $d_j$.

In the remainder of this chapter, we make the following assumptions: (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 $M_p \le N\lfloor \frac{k}{p} \rfloor$, since at most $\lfloor \frac{k}{p} \rfloor$ items of size $p$ can be stored on a single node.

In [70, 156] 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 node systems [70]. In this work, we generalize this problem to the case where we can have non-uniform sized data items. For the previous algorithms [70, 156] 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 [156], 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 node. 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 $k \ge 2$. 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 $(1 - \frac{1}{(1+\sqrt{\lfloor k/\Delta \rfloor})^2})$ (easy to check that this would be a tight bound). Our results for items of size 1 and 2 suggests that such a bound should hold for any value $\Delta$.

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 (ratio of $|U_i|/S_i$) 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 $\Delta$, we develop a polynomial time approximation scheme (PTAS) as follows. For a given $\epsilon > 0$, if $(1 - \epsilon) \le f(k, \Delta)$ 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 $\epsilon$ and $\Delta$) 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 [70], 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 nodes. For any fixed integer $k, \Delta$ and $\epsilon > 0$ this algorithm runs in polynomial time and outputs a solution where at least $(1 - \epsilon)$-fraction of the clients in an optimal solution are assigned.

At this point, it is worth noting that while there is a PTAS for the problem for a constant number of distinct sizes (Section 2.7 of this chapter, and the independent work in [157]), *even for the simplest case* when the data items have unit sizes (for example the first PTAS in [70]), none of the approximation schemes are actually practical since the running times are too high, albeit polynomial for a fixed $\epsilon$. The only known algorithms that are practical, are the ones based on the sliding window approach. Hence even though the bounds that one can derive using sliding window based methods can be improved by other approaches, this still remains the best approach to tackling the problem from a practical standpoint. Obtaining a practical PTAS remains an outstanding open problem.

### 2.1.1 Related Work.

The data placement problem described above bears some resemblance to the classical multi-dimensional knapsack problem [143, 29]. However, in our problem, the storage dimension of a node behaves in a *non-aggregating* manner in that assigning additional clients corresponding to a data item that is already present on the node 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 [156] 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 [156] showed that the problem is NP-hard when each node has an arbitrary load capacity, and unit storage. Golubchik *et al.*[70] 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 (under certain assumptions) to pack a $(1 - \frac{1}{(1+\sqrt{k})^2})$-fraction of items for any instance of identical nodes. Moreover, there exists a family of instances for which it is infeasible to pack any larger fraction of items. The problem with identical nodes is shown to be NP-hard for any fixed $k \geq 2$ [70].

In addition, packing problems with color constraints are studied in [47, 155]. 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 there is a constraint on the number 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.

Independently, Shachnai and Tamir [157] have recently announced a result similar to the one presented in Section 2.7. For any fixed $\epsilon$ and a constant number of sizes $s_i \in \{a_1, \ldots, a_c\}$ and for identical parallel nodes they develop a polynomial time approximation scheme where the running time is polynomial in $N$ and $M$, the number of nodes and data items. Since this does not assume constant $k$, they do not need a separate algorithm when $k$ is large. However, the algorithms and the ideas in their work are based on a very different approach as compared to the ones taken in this chapter.

### 2.1.2 Other Issues.

Once a placement of items on the nodes has been obtained, the problem of assigning clients to nodes 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 nodes, 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.

### 2.1.3 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 nodes are a particularly interesting resource. Firstly, nodes 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 node resources are not equivalent since a node'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 [20]. However, this approach suffers from a number of implementation-related shortcomings that are detailed in [33]. An alternative system is described in [172] where the nodes are connected in a shared-nothing manner [165]. 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 this chapter 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 [172], 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.

### 2.1.4 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 the nodes. Note that this function is always at least $\frac{3}{4}$ and approaches 1 as $k$ goes to $\infty$. When data items have sizes $s_i \in \{1, 2, 2^2, \ldots, \Delta\}$ where $\log \Delta \in Z$, we generalize this multi-list sliding window algorithm to guarantee that at least $(1 - \frac{1}{(1+\sqrt{\lfloor \frac{k}{\Delta} \rfloor})^2})$-fraction of clients will be assigned to the nodes. Moreover, we can show that these bounds are 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 node, when $s_i \in \{1, \ldots, \Delta\}$.

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 (PTAS) that takes as input parameter two constants $k$ and $\epsilon'$ and yields a $(1-\epsilon')^3$ approximation to the optimal solution, in time that is polynomial for fixed $k$ and $\epsilon'$. Pick $\epsilon'$ 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 PTAS with parameters $\epsilon'$ and $k$, both of which are constant for fixed $\epsilon$. This gives a polynomial time approximation scheme.

## 2.2 Sliding Window Algorithm

For completeness we describe the algorithm [156] that applies to the case of identical nodes with unit size items.

At step $j$, we assign items to node $d_j$. 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 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, $R[1], \ldots, R[m]$, $1 \leq m \leq M$, is updated during the algorithm. At first, $m = M$ and $R[i] = |U_i|$. 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 node $d_j$ has the general rule that we want to select the first consecutive sequence of $k$ or less data items, $R[u], \ldots, R[v]$, whose total number of clients is at least the load capacity $L$. We then assign items $R[u], \ldots, R[v]$ 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 node 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 $R[2], \ldots, R[k+1]$ 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.

## 2.3 Multi-List Sliding Window Algorithm for $\Delta = 2$

The proof of the tight bound in [70] involves obtaining an upper bound on the number of data items that were not packed in any node, 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 node simply because we are left with only size 2 items and cannot pack them perfectly.

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 $R_1$, $R_2$ 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 $R_1$ and $R_2$ respectively. The top $N$ size-1 items with the *highest demand* are moved into *aux-list*. The remaining size-1 items are kept in $R_1$. All the size-2 items are placed in the $R_2$ list. From this stage on, the algorithm maintains the $R_1$, $R_2$ and *aux-list* lists in non-decreasing order of demand.

For each node (stage), the algorithm must make a selection of items from $R_1$, $R_2$ 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. 2.3.1.2). The algorithm then selects $w_1$ consecutive items from $R_1$ and $w_2$ consecutive items from $R_2$ such that the total utilized space of the selected items from $R_1$ and $R_2$ 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 $R_2$, one from $R_1$ 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 $R_2$ 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 $R_1$ 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 $1 \ldots |$*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 $R_1$ 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 by moving one item from *aux-list* into $R_1$. The size-1 item that leaves *aux-list* in the previous step is then reinserted into the appropriate position of the $R_1$ 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.

### 2.3.1 Analysis of the Algorithm

For each node 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 node) for each of the items assigned to the node. We will argue that the ratio of packed demand 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. 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. We denote the number of packed clients by S and the number of unpacked clients by U.

#### 2.3.1.1 Even K.

Given an instance $I$ create a new instance $I^{'}$ by merging arbitrary 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 2.2. The basic idea is to view a capacity $k$ node as a capacity $k/2$ since each item has size 2. From the result of [70], 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.

#### 2.3.1.2 Odd K.

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

We will first bound the space wasted in packing the load-saturated nodes and then bound the space wasted in packing the non load-saturated nodes 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 combination of windows per stage which corresponds to the assignment for a single node. 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

$R_1$ is the list of $(M_1 - N)$ size-1 items, $R_2$ 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 $R_1$ and $R_2$ 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 nodes, they will be packed completely (this actually follows from [156], see [70] for a simpler proof). Furthermore it is not hard to show that if at any stage $j$, we have produced $j-1$ load-saturated nodes and the total size of the objects in the $R_1$ and $R_2$ lists is $\leq k-1$, then all the items will be packed at the termination of the algorithm. The running time of this algorithm is $O(n^4k^3)$.

**Lemma 2.3.1.** *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

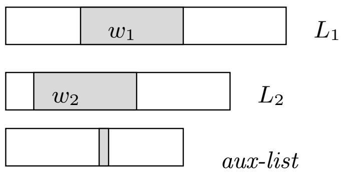


**Figure 2.1:** Lists used by Algorithm.

a load-saturated node. 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 $R_1$, $m_2^1, \ldots, m_2^j$ from $R_2$ ($j = 0$ implies $w$ has no items from $R_2$) 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 $R_1$ ($i^{'} = 0$ implies $w^{'}$ has no items from $R_1$), $m_2^{j+1}, \ldots, m_2^{j+j^{'}}$ from $R_2$ ($j^{'} = 0$ implies $w^{'}$ has no items from $R_2$) and *aux-item(2)* from *aux-list* (this item is mandatory). Let $\ell_q^p$ 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 the 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 the load is at most $L$ and we keep adding items from $R_2$ and dropping items from $R_1$, 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 $m_2^{j+1}$.

□

**Lemma 2.3.2.** *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 $\geq L$*

*Proof.* Assume not. Now $w$ (the current window) has items say $m_1^1, \ldots, m_1^i$ from $R_1$ ($i = 0$ implies $w$ has no items from $R_1$), $m_2^1, \ldots, m_2^j$ from $R_2$ ($j = 0$ implies $w$ has no items from $R_2$) and *aux-item(1)* from *aux-list* (this item is mandatory). Consider a leftmost window (call it $w'$) with the same size as $w$ and with load $< L$. Also $w'$ has items say $m_1^1, \ldots, m_1^{i'}$ from $R_1$ ($i' = 0$ implies

$w^{'}$ has no items from $R_1$), $m_2^1, \ldots, m_2^{j^{'}}$ from $R_2$ ($j^{'} = 0$ implies $w^{'}$ has no items from $R_2$) and *aux-item(1)* from *aux-list* (this item is mandatory). Since $w^{'}$ has the same size as $w$ but is different from $w$, one of the following must be true:

1. $j' < j$. Since $size(w^{'}) \leq k - 2$, add in the items from $R_2$ starting from $m_2^{j^{'}} + 1$ until $size(w^{'}) = k$ or until the load of $w^{'}$ becomes $> L$. If the load of $w^{'}$ becomes $> L$ and we have managed to add in an item, then we have a contradiction since we have found a window larger than $w$ that is load-feasible within one split. Note that if we add in items upto $m_2^j$, the load of $w^{'}$ must become $> L$ and as before if we have managed to add in an item, then we have a contradiction. So now, we have $size(w^{'}) = k$ and the load of $w^{'}$ is $<$ L and we have not yet added in $m_2^j$. Now we drop the two highest items in $w^{'}$ from $R_1$ and add in the next higher item (not already in $w^{'}$) from $R_2$ and repeat until we have either added in $m_2^j$ or until the load of $w^{'}$ becomes $> L$. In either case, we have a contradiction since we have found a larger feasible window than the current window.

2. $j' > j$. Since $size(w^{'}) \leq k - 2$, add in the items from $R_1$ starting from $m_1^{i^{'}} + 1$ until $size(w^{'}) = k$ or until the load of $w^{'}$ becomes $> L$. If the load of $w^{'}$ becomes $> L$ and we have managed to add in an item, then we have a contradiction since we have found a window larger than $w$ that is load-feasible within one split. Note that if we add in items upto $m_1^i$, the load of $w^{'}$ must become $> L$ and as before if we have managed to add in an item, then we have a contradiction. So now, we have $size(w^{'}) = k$ and the load of $w^{'}$ is $<$ L and we have not yet added in $m_1^i$. Now we drop the highest item in $w^{'}$ from $R_2$ and add in the next higher items (not already in $w^{'}$) from $R_1$ and repeat until we have either added in $m_1^i$ or until the load of $w^{'}$ becomes $> L$. In either case, we have a contradiction since we have found a larger feasible window than the current window.

□

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

**Lemma 2.3.3.** *If at the termination of the algorithm there are unassigned clients then for every load-saturated node $d_i$ one of the following conditions must hold:*

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

2. *Node $d_i$ has $us(i) = k$ and a size-2 item is split.*

*Proof.* We need to show that if we produce a load-saturated node that violates conditions (1) and (2) then all the items from all the lists ($R_1$, $R_2$ and *aux-list*) will be packed completely.

From Lemma 2.3.1, 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 nodes.

From Lemma 2.3.2 we know that when the current window has $\geq 2$ 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 nodes until the total load of the items remaining on $R_1$ and $R_2$ is $< L$. However the total size of the items remaining on $R_1$ and $R_2$ 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 2.3.4.** *If at the termination of the algorithm there are unassigned clients then either*

1. *All the non load-saturated nodes are size-saturated.*

2. *Only size-2 items are remaining and there is at most one non load-saturated node with exactly one unit of unused space and all the other non load-saturated nodes are size-saturated.*

*Proof.* If at the termination of the algorithm, $R_1$ is not empty then all the non load-saturated nodes *must* also be size-saturated; otherwise the algorithm would have found a selection with higher load by adding in another item from $R_1$.

Now consider the case where $R_1$ is empty and $R_2$ is not empty. Since $R_2$ is not empty, for each non-load saturated node $i$ we have $us(i) \geq k-1$. Now assume (for contradiction) that there are two non load-saturated nodes $i$ and $j$ (say $i < j$) s.t. $us(i) = us(j) = k-1$. If we have $us(i) = k-1$ then $R_1$ must become empty after this selection has been assigned to $i$; otherwise, the algorithm could just have added in another item from $R_1$ and would have found a selection with higher load. Since $us(i) = k-1$, the $R_1$ list becomes empty after the current selection has been assigned to node $i$. Now $R_1$ is empty and exactly one item from *aux-list* will be forced onto $j$, so for all future nodes $j > i$, $us(j)$ 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.

$\square$

**Theorem 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 2.3.3 and 2.3.4, 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 theorem 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 below to upper bound $\frac{U}{S}$.

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.

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

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

$$\begin{aligned}
\frac{cy}{\lfloor \frac{k}{2} \rfloor} &= (1-c)(1-y) \\
y &= \frac{1-c}{1-c+\frac{c}{\lfloor \frac{k}{2} \rfloor}}
\end{aligned}$$

Substituting for $y$ gives us that the upper bound for $U/S$ is at most $\frac{c-c^2}{\lfloor \frac{k}{2} \rfloor - \lfloor \frac{k}{2} \rfloor c + c^2}$. This achieves its maxima when $c = (1 - \frac{1}{1+\sqrt{\lfloor \frac{k}{2} \rfloor}})$. The fraction of all the items that are packed is

$$\begin{aligned}
\frac{S}{U+S} &= \frac{1}{1+\frac{U}{S}} \qquad (2.1) \\
\frac{S}{U+S} &\geq (1 - \frac{1}{(1+\sqrt{\lfloor k/2 \rfloor})^2})
\end{aligned}$$

We now prove that $U \leq \frac{N_l cL}{\lfloor \frac{k}{2} \rfloor}$:

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 node be $cL$. Since any non load-saturated node 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 $\leq \frac{cL}{\lfloor \frac{k}{2} \rfloor}$ 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 $\leq \frac{N_l cL}{\lfloor \frac{k}{2} \rfloor}$.

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 nodes have $k$ units of utilized space. This node 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 nodes 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 node. Therefore, the maximum load of the smallest object on the lightest loaded node is $\leq \frac{cL}{\lfloor \frac{k}{2} \rfloor + 1}$. The load of the single size-1 item on the remaining items list must be at most $x$ and must also be $\leq \frac{cL}{\lfloor \frac{k}{2} \rfloor + 1} \leq \frac{cL}{\lfloor \frac{k}{2} \rfloor}$ 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 node.

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

   - If $m_1^{'} \geq 2$. 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 $m_1^{'} \geq 2$, 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 node. 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 $i \neq j$. 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

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

Thus

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

Simplifying yields

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

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

□

## 2.4 Tight Algorithm for $s_i \in \{1, 2, 2^2, \ldots, \Delta\}$

We extend the multi-list algorithm from Section 2.3 for the case when sizes are in the set $S = \{1, 2, 2^2, \ldots, \Delta\}$ and $\log \Delta \in Z$. Say $k = p\Delta + q$ where $p, q, \Delta \in Z$ and $0 \leq q < \Delta$. The approach, presented in this section, to obtain a tight algorithm for an instance where $S = \{1, 2, 2^2, \ldots, \Delta\}$ is to first reduce it to an instance where $S = \{q, \Delta\}$ and then extend the algorithm from Section 2.3 to solve the simpler instance. As in Section 2.3, we assume that *ignoring load constraints*, there exists a feasible packing of all the items. Also, as earlier, we assume that the total size of all the items in the input instance is equal to $Nk$ (inserting dummy size-1 items with zero load, if necessary, to ensure this) and that the total demand of all the items in the input instance is equal to $NL$.

### 2.4.1 Reduction to $S = \{q, \Delta\}$

Given an instance where $S = \{1, 2, 2^2, \ldots, \Delta\}$, we will reduce it to an instance where $S = \{q, \Delta\}$ so that if there was a packing of all the items (ignoring load constraints) in the original instance, then there will be a packing of all the items (ignoring load constraints) in the reduced instance.

To ensure that the reduced instance has a packing of all the items (ignoring load constraints), we will create $N$ groups of size $q$ each and $Np$ groups of size $\Delta$ each. To verify that such groups must exist in any instance where $S = \{1, 2, 2^2, \ldots, \Delta\}$ and where all the items can be packed (ignoring load constraints), consider any such packing that assigns all the items. Now on each disk we form as many size-$\Delta$ groups using these items as possible. This is done by consecutively merging groups of the same size smaller than $\Delta$ on each disk, until on each disk there exists at most one group each of any size $< \Delta$ along with some number of groups of size $\Delta$. These remaining groups with size $< \Delta$, at most one of each such size, are then merged into a single group of size say $g$. Note that $g < \Delta$. Since $k = p\Delta + q$ and since each disk was packed to its full capacity $k$, there must be $p$ groups of size $\Delta$ and one group of size $g = q$ on each disk. Also note that $q$ is the sum of a unique subset of sizes from $S$. For instance, $q = 7$ can only be represented (without repetitions) as $q = 1 + 2 + 4$ if $S = \{1, 2, 4, 8, \ldots, \Delta\}$. Say $q = \sum_{i=1}^{r'} a_i$ where $a_i \in A \subseteq S/\Delta$. Any group of size $q$ can therefore be further broken down into $r' = |A|$ groups of size $a_i$ each. So there must be $N$ groups of size $a_i$ each for every $a_i \in A$ and $Np$ groups of size $\Delta$ each. So as long as we show how to form these groups, we have shown that there is a packing of all the items (ignoring load constraints) in the reduced instance.

We form these groups as follows. Let $m(s_i)$ denote the number of size $s_i$ items for any $s_i \in S$. For any $i, j$ let $s_i < s_j$ if $i < j$. We have $s_j = 2^{j-1}$ for any $1 \leq j \leq \log \Delta + 1$. Let $s_0 = 0$ and $m(s_0) = 0$. The initial number of items is then represented by $\{m(1), m(2), m(2^2), \ldots, m(\Delta)\}$. The algorithm then proceeds in rounds starting from round $i = 1$ and ending when round $i = r'$ is complete.

At the end of round $i - 1$ (for $i > 1$) there must be at least $N$ groups of size $a_{i-1}$ (see Lemma 2.4.1). Before the start of round $i (> 1)$, the algorithm ensures that there are exactly $N$ groups of size $a_{i-1}$. Say $m(a_{i-1}) = N + n'$. If $n' > 0$, we pair up each of the $n'$ items to form $\lfloor n'/2 \rfloor$ groups of size $s_j = 2a_{i-1}$. Before round $i = 1$, we start with zero items of size $a_0 = 0$. In round $i$, we form at least $N$ groups of size $a_i$. All items with sizes $s_j$ such that $a_{i-1} < s_j < a_i$ are merged together to form size $a_i$ groups as follows. Starting from $s_j = 2 * a_{i-1}$ two items with size $s_j$ are merged together to form an item of size $s_{j+1}$. This merge operation decreases $m(s_j)$ by 2 and increases $m(s_{j+1})$ by 1. Once $m(s_j) \leq 1$, we continue this process with size $s_{j+1}$ items. The process stops when $m(s_j) \leq 1$ where $s_{j+1} = a_i$. This way we have merged all items smaller than $a_i$ but larger than $a_{i-1}$ into size $a_i$ groups. This marks the end of round $i$.

**Lemma 2.4.1.** *At the end of round $i$ (for $i \geq 1$) the algorithm produces at least $N$ groups of size $a_i$ where $i \leq r^{'}$, $q = \sum_{i=1}^{r^{'}} a_i$ and $a_i \in A \subseteq S/\Delta$.*

*Proof.* (Sketch) We have the invariant that before round $i$ begins, there are a maximum number of size $s_k = 2 * a_{i-1}$ groups that can be formed using the instance excluding $N$ groups of each size $a_j$, $j \leq i-1$.

This is trivially true before round $i = 1$ begins. To see that the invariant holds true before the start of the next round $i+1$, consider round $i$. In round $i$, starting with $s_j = 2 * a_{i-1}$, the algorithm forms as many groups of size $s_{j+1}$ as possible (excluding $N$ groups of each size $a_j$, $j \leq i-1$) by merging pairs of groups of size $s_j$. This merging results in the largest possible number of size $s_{j+1}$ groups (excluding $N$ groups of each size $a_j$, $j \leq i-1$) since any group of size $s_{j+1}$ can *only* be formed using either two groups of size $s_j$ or using a single item of size $s_{j+1}$ and we start with the maximum possible number of size $s_j$ items (excluding $N$ groups of each size $a_j$, $j \leq i-1$). So at the end of round $i$, the algorithm creates as many size $s_j = a_i$ groups as possible excluding $N$ items of each size $a_j$, $j \leq i-1$.

Since there exists a packing of all the items (ignoring load constraints) we know that there are at least $N$ groups of each size $a_j$ for $j \leq i$ and we have the maximum possible number of size $a_i$ groups (excluding $N$ items of each size $a_j$, $j \leq i-1$) at the end of round $i$. Consequently $m(a_i)$ must be at least $N$. Say $m(a_i) = N + n^{'}$. If $n^{'} > 0$, we pair up each of the $n^{'}$ items to form $\lfloor n^{'}/2 \rfloor$ groups of size $s_{j+1} = 2a_i$. Again, we have the maximum number of size $s_{j+1} = 2a_i$ groups along with exactly $N$ groups of each size $a_j$, $j \leq i$ and the invariant holds before the start of round $i+1$. Consequently at the end of round $r^{'}$ the algorithm produces at least $N$ groups of size $a_i$ where

$i \leq r^{'}$. □

At the end of round $r^{'}$, we have $m(a_i) = N + n^{''}$ (see Lemma 2.4.1). As before if $n^{''} > 0$, we pair up each of the $n^{''}$ items to form $\lfloor n^{''}/2 \rfloor$ groups of size $s_j = 2a_{r'}$. If $s_j = \Delta$, we are done. If not, starting from $s_j = 2a_{r'}$ two items with size $s_j$ are merged together to form an item of size $s_{j+1}$. Once $m(s_j) \leq 1$, we continue this process with size $s_{j+1}$ items. The process stops when $m(s_j) \leq 1$ where $s_{j+1} = \Delta$.

**Lemma 2.4.2.** *At this stage of the algorithm, there are exactly $N$ groups of each size $a_i$ and zero groups of any size $s_j \neq a_i$ where $s_j < \Delta$ and $1 \leq i \leq r^{'}$.*

*Proof.* We know from Lemma 2.4.1 that there must be at least $N$ groups of size $a_i$ for each $1 \leq i \leq r^{'}$. The algorithm ensures that there is at most one remaining group of any size $s_j$ where $s_j < \Delta$, excluding $N$ groups of each size $s_j = a_i$ for each $1 \leq i \leq r^{'}$ (otherwise it would have found another pair of items to merge). The sum of the sizes of all these remaining groups is some $g < \Delta$ since there is at most one each of any size $< \Delta$. Now since the original instance had $N$ groups of size $a_i$ each and $Np$ groups of size $\Delta$ it follows that since we already have $N$ groups of size $a_i$ each, the total space of the size $\Delta$ groups and the size $g$ group must be equal to $Np\Delta$. However, since $g < \Delta$ and all the other groups have size $= \Delta$, it follows that we must have that $g = 0$. □

Note that while the merge operation changes the number of groups in the instance, the total size of the groups remains unchanged. So at the end of this final round, since the total space of items used to form these groups is $Np\Delta$ (this follows from Lemma 2.4.2) and because we form groups of size exactly equal to $\Delta$, we must have $Np$ groups of size $\Delta$. We complete the reduction by forming $N$ groups of size $q$ each by merging one group of each size $a_1, \ldots, a_{r'}$.

### 2.4.2 Tight algorithm for the reduced instance

The algorithm works in two phases. In the first phase, it attempts to pack all the items using an algorithm similar to that of Section 2.3. In the second phase, the algorithm improves this packing using local-search swaps.

During the first phase, the algorithm has two lists; $q$-list consisting of groups of size $q$ ($N$ initially) arranged in non-decreasing order of demand and $\Delta$-list consisting of groups of size $\Delta$ ($Np$ initially) arranged in non-decreasing order of demand.

For each node, the algorithm must make a selection of groups from $q$-list and $\Delta$-list. Exactly one group is always chosen from $q$-list and $w_1$ ($1 \leq w_1 \leq p$) consecutive groups from $\Delta$-list are chosen.

Define wasted space of a selection to be the sum of the unused space and the size of the size of the group that must be split to make the selection load-feasible. At each stage, the algorithm makes a list of selections $\mathcal{L}$ by combining the following selections - $w_1$ ($1 \leq w_1 \leq p$) consecutive groups from $\Delta$-list and one size-q group from $q$-list.

If $\forall l \in \mathcal{L}$, load($l$) $< L$ the algorithm outputs the selection with highest load. If $\exists l \in \mathcal{L}$ where load($l$) $\geq L$, then let $D$ be the set of all the selections in $\mathcal{L}$ with load $\geq L$. Let $D^{'} \subseteq D$ be the set of all the selections which can be made load-feasible by allowing the split of either the highest size-$\Delta$ group or the size-q group in the selection. The algorithm choses $d \in D^{'}$ 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 group $d_i$. If $d_i^{''} > 0$ the algorithm reinserts $d_i^{''}$ into the appropriate position in the list form which $d_i$ was chosen. If us($i$) $< k$, then the split piece is forced into the next selection. The algorithm proceeds in this way until its outputs a selection of groups for each of the $N$ nodes.

**Lemma 2.4.3.** *If us(i) $< k$ for any load-saturated node, then all the demand is packed when the algorithm terminates*

*Proof.* Suppose the items assigned to a load saturated node $i$ are $x_1, \ldots, x_r$ (in non-decreasing order of load). Then $\sum_{j=1}^{r-1} l_j < L$ and $\sum_{j=1}^{r} l_j = L$. Note that the items must be the first $r-1$ items from the $\Delta$-list along with the first item from the $q$-list. If not, then we can derive a contradiction by finding a selection with load $\geq L$ but with lesser wasted space. Note that if us($i$) $< k$ then us($i$) $\leq k - \Delta$. As a result, the split piece can always be accommodated in the selection for the next round. Further, every future selection with us($i$) $\geq k - \Delta$ has load $\geq L$. So the algorithm continues to produce load-saturated nodes until all items are packed or until we run out of nodes. □

**Lemma 2.4.4.** *At the end of phase 1 of the algorithm there are at most $N_l$ groups that remain unpacked.*

*Proof.* Call the packing we obtain at the end of phase 1 as $P$. To count the number of unpacked groups in $P$, note that we assume there is a packing of all items (ignoring load constraints) in the original instance. Consider any such packing say $P^{'}$. The total space of the groups in this packing $P^{'}$ is $Nk$. In our packing $P$, only a single group is split on any load-saturated node and consequently at most $N_l$ new copies of groups are created. The newly created groups have size identical to one of the original groups. Further, the total space of packed groups in packing $P$ from Lemma 2.4.3 is $Nk$. Consequently, the packing at the end of phase 1, $P$ can be transformed into packing $P^{'}$ by a series of swaps between groups in $P$ and the list of unpacked groups. Note that the swaps do not change the number of unpacked groups, they simply change which groups were not packed. Consequently, once we transform the packing at the end of phase 1, $P$ into the packing $P^{'}$, the only groups that remain in the list of unpacked groups are the newly created groups and there are at most $N_l$ such groups. □

If there are no size-q items remaining at the end of phase 1, then we are done. However, if there are size-q items remaining at the end of phase 1, then we need to do local-search swaps to improve the packing. These local-search swaps are performed during phase 2.

In phase 2, if the size-q list is not empty, the size-q groups are first broken into their constituent items. Recall that in Section 2.4.1, an instance where sizes were originally in the set $S = \{1, 2, 2^2, \ldots, \Delta\}$ was reduced to an instance where sizes are in the set $S = \{q, \Delta\}$. The algorithm then forms as many size-$\Delta$ groups using these items as possible. This is done, as in Section 2.4.1, by consecutively merging groups of the same size smaller than $\Delta$, until there exists at most one group each of any size $< \Delta$ along with some number of groups of size $\Delta$. These remaining groups with size $< \Delta$, at most one of each such size, are then merged into a single group, say this group has size $g$. Note that $g < \Delta$. At the end of this merge operation, the algorithm produces some number of size-$\Delta$ items along with at most one size-g group. The algorithm places all these groups (including the size-g group) in the remaining groups list $R$. The list $R$ now has these newly created groups along with the groups that remained unpacked at the end of phase 1. Note that $R$ has at most $N_l$ groups since the merge operation cannot increase the number of unpacked groups and from Lemma 2.4.4, there were at most $N_l$ unpacked groups before the start of the merge operation.

The algorithm then does a series of local-search swaps to improve the load of the packing. In the local-search step, the least profitable group on a non load-saturated node is swapped out and the most profitable group remaining in $R$ is swapped in. If the swap results in an improvement in the load of the node, then the swap is retained else no improvement is possible and the phase ends. If the swap results in an improvement and also causes the node to become load-saturated, then the item that was swapped in is split (as before) to make the node load-feasible and the remaining piece is reinserted into the remaining items list $R$. Note that the local-search operation guarantees that the number of items in $R$ never exceeds $N_l$. This is because a new item is added to $R$ only if a previously non load-saturated node becomes load-saturated and hence $N_l$ increases by 1. Consequently, we have the following Lemma.

**Lemma 2.4.5.** *At the end of phase 2 of the algorithm, there are at most $N_l$ groups that remain unpacked.*

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

*Proof.* As a result of the Lemmas 2.4.4 and 2.4.5, we know that at the termination of the algorithm there are at most $N_l$ unpacked groups. The local-search swaps during phase 2 ensure that the load any unpacked group is at most the load of the least profitable group on a non-load saturated disk. Let the load on the lightest loaded non load-saturated node be $cL$. Since any non load-saturated node must have at least $\lfloor \frac{k}{\Delta} \rfloor + 1$ groups, any unpacked group can therefore have load at most $\frac{cL}{\lfloor \frac{k}{\Delta} \rfloor + 1} < \frac{cL}{\lfloor \frac{k}{\Delta} \rfloor}$. The total load of the remaining items ($U$) is therefore $\leq \frac{N_l cL}{\lfloor \frac{k}{\Delta} \rfloor}$.

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.

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

Since

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

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

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

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

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

$$\begin{aligned} \frac{U}{S} &\leq \frac{\min(\frac{cy}{\lfloor \frac{k}{\Delta} \rfloor}, (1-c)(1-y))}{y + c(1-y)} \\ \frac{U}{S} &\leq \min(\frac{\frac{cy}{\lfloor \frac{k}{\Delta} \rfloor}}{y + c(1-y)}, \frac{(1-c)(1-y)}{y+c(1-y)}) \end{aligned}$$

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

$$\begin{aligned} \frac{cy}{\lfloor \frac{k}{\Delta} \rfloor} &= (1-c)(1-y) \\ y &= \frac{1-c}{1-c+\frac{c}{\lfloor \frac{k}{\Delta} \rfloor}} \end{aligned}$$

Substituting for $y$ gives us that the upper bound for $U/S$ is at most $\frac{c-c^2}{\lfloor \frac{k}{\Delta} \rfloor - \lfloor \frac{k}{\Delta} \rfloor c + c^2}$. This achieves its maxima when $c = (1 - \frac{1}{1+\sqrt{\lfloor \frac{k}{\Delta} \rfloor}})$. The fraction of all the items that are packed is

$$\begin{aligned} \frac{S}{U+S} &= \frac{1}{1+\frac{U}{S}} \\ \frac{S}{U+S} &\geq (1 - \frac{1}{(1+\sqrt{\lfloor \frac{k}{\Delta} \rfloor})^2}) \end{aligned} \tag{2.2}$$

□

## 2.5 Tight Example for $s_i \in \{1 \ldots \Delta\}$

The tight example in this section is for the case where $s_i \in \{1, \ldots, \Delta\}$ and is an extension of the tight example presented in [70] for the uniform size sliding window algorithm. The tight example instances will only consist of size-$\Delta$ items.

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

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

First consider the $\sqrt{\lfloor \frac{k}{\Delta} \rfloor}$ large items. An unsplit item $U_i$ has all its demand allocated to a single node. A split item $U_i$ has its demand allocated to several nodes. For a node that contains at least one large unsplit item, the available load capacity is at most $\lfloor \frac{k}{\Delta} \rfloor - 2$. 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 node, 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 node. The nodes now have one large unsplit item and at most $\lfloor \frac{k}{\Delta} \rfloor - 2$ small items. The remaining nodes have only large split items. Assume that there are exactly $p (0 \leq p \leq \lfloor \frac{k}{\Delta} \rfloor)$ large items that do not get split $U_1, \ldots, U_p$ with node $d_i$ containing $U_i$.

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

## 2.6 Generalized Sliding Window Algorithm (SW-Alg2)

The sizes of the items in our instance are chosen from the set $\{1, \dots, \Delta\}$. In this section, we present algorithm SW-Alg2 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 $N$ nodes each with storage capacity $k+\Delta-1$ and load capacity $L$. In the second phase, the algorithm makes the solution feasible by dropping items from these nodes until the subset of items on each node has size at most $k$.

In the first phase of the algorithm, the algorithm keeps the items in a list sorted in non-decreasing order of density $\rho_i$, 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 node, 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 $k+\Delta-1$. 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 $x_u, \dots, x_v$. We have $\sum_{i=u}^{v} l_i \geq L$, $\sum_{i=u}^{v-1} l_i < L$, $\sum_{i=u}^{v} s_i \leq k+\Delta-1$ 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 $s_{p-1} + \sum_{i=p}^{q} s_i > k + \Delta - 1$ (if $x_p$ is not the first item in the list of remaining items) and $\sum_{i=p}^{q} s_i \leq k + \Delta - 1$. 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 node, 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) $\leq k + \Delta - 1$, then add $x_i$ to current selection. Else if $load$(current selection) $< L$ and $s_i + size$(current selection) $> k + \Delta - 1$, drop the lowest density items from current selection as long as $s_i + size$(current selection) $>$ $k + \Delta - 1$, 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 node 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 node, the algorithm drops the lowest density items assigned to the node 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 3.** *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 2.6.1.** *If $us(i) \leq k - \Delta$ for any load-saturated node $i$ 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 node $i$ are $x_1, \ldots, x_p$ (in non-decreasing order of density). Then we have $\sum_{j=1}^{p} s_j \leq k - \Delta - 1$, $\sum_{j=1}^{p} l_j \geq L$ 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 $size(\text{current selection}) > k + \Delta - 1$. The algorithm will then drop items to make the current selection size feasible. Since each item has size at most $\Delta$, 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 $\leq k - \Delta - 1$ 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 $k - 1$ has load $\geq L$. Since the algorithm permits every node to pack items of total size upto $k + \Delta - 1$ in phase I, note that the broken off piece (which has size $\leq \Delta$) from the previous load-saturated node can be accomodated in the next load-saturated node. In this way, the algorithm produces load-saturated nodes until there are no more items in the remaining items list. □

**Lemma 2.6.2.** *At the end of phase I of the algorithm, at least a $\left(1 - \frac{1}{\left(1+\sqrt{\frac{k}{2\Delta}}\right)^2}\right)$-fraction of clients are 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 nodes in the assignment and $cL$ denotes the load of the lightest loaded non load-saturated node in the assignment. The bound will then follow from the method outlined in the proof of Theorem 1. (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 node. This is because when we are unable to produce any more load-saturated nodes, 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 node. Since the load of the lightest loaded node is $cL$, we also have $\sum_{i=1}^{p} load(x_i) = cL$. Since in phase I we allow each node to be filled upto size $k + \Delta - 1$, we have $k < \sum_{i=1}^{p} size(x_i) < k + \Delta$, unless all the items have been packed. Let $\rho_{min}$ denote the density of the lowest density item assigned to the lightest loaded node. 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 $\rho_{min}$, 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 2.6.1, we know that each load-saturated node is filled to size $> k - \Delta$. Further, we know that each non load-saturated node is filled to size $> k$, otherwise we can add an item to this node. Since our instance was feasible, the total size of all the items in the instance is $Nk$. Every time we create a load-saturated node we might split at most one item and this item can have size at most $\Delta$. 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 nodes and $N_s$ is the number of non load-saturated nodes. 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 2.6.3.** *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 node 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 $\Delta$, $\sum_{j=1}^{q} size(x_j) \leq 2\Delta$ and $\sum_{i=q+1}^{p} s_i > k - \Delta$. Let $\rho$ 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 node we can lower bound the remaining load (after dropping items $x_1, \ldots, x_q$ to make it size-feasible) as follows:

$$\textstyle\sum_{i=q+1}^{p} \rho_i s_i \quad \geq \quad \rho \sum_{i=q+1}^{p} s_i \quad > \quad \rho\,(k - \Delta)$$

Further, for each node 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 2.1). $\square$

Using these two lemmas, we easily obtain the proof of Theorem 3.

## 2.7 Polynomial Time Approximation Schemes

From Theorem 3 we know that when the sizes of our items are chosen from the set $\{1 \ldots \Delta\}$, algorithm SW-Alg2 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. Say $f(k, \Delta) = \frac{k-\Delta}{k+\Delta}\left(1 - \frac{1}{\left(1+\sqrt{\frac{k}{2\Delta}}\right)^2}\right)$.

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 [70]. Algorithm PTAS takes as input parameters $k, c$ and $\epsilon'$ 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 $\Delta$, 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 $\epsilon' \leq \frac{1}{k}$. If this is not the case, we simply lower the value of $\epsilon'$ to $\frac{1}{k}$. Since $k$ is a fixed constant, lowering the value of $\epsilon'$ only yields a better solution, and the running time is still polynomial.

The approximation scheme involves the following basic idea:

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 nodes.

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 $\alpha$ to denote $1/\epsilon'$. Also, for any solution $S$, we use $|S|$ to denote the number of items packed by it.

### 2.7.1 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 the same as in [70].

**Lemma 2.7.1.** *For any instance $I$, we can construct in polynomial time another instance $I'$ such that*

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

### 2.7.2 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 node 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 nodes and popular data items and between heavy nodes and unpopular data items, obeys some nice properties. The proof of the following lemmas is in [70].

**Lemma 2.7.2.** *For any instance $I$, there exists a solution $S$ satisfying the following properties:*

- *at most one light node receives clients from a set $U_j$.*
- *a heavy node 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 node is said to be $\delta$-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 $\beta$ is a non-negative integer.

**Lemma 2.7.3.** *Any solution $S$ can be transformed into a solution $S'$ such that*

- *each heavy node in $S$ is $(\epsilon'^2/k)$-integral in $S'$ w.r.t. each popular data item, and*
- *$S'$ packs at least $(1-\epsilon')|S|$ items.*
- *each heavy node packs $(1-\epsilon')L$ items corresponding to popular items.*

### 2.7.3 The Approximation Scheme.

Start by preprocessing the given input instance $I$ so as to create an $(\alpha L)$-bounded instance $I'$ as described in Lemma 2.7.1. We now give an algorithm to find a solution $S$ to $I'$ such that $S$ satisfies the properties described in Lemmas 2.7.2 and 2.7.3 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 nodes in $O$, say $N'$. Let $\mathcal{H}$ be the set of nodes $d_1$ through $d_{N'}$ and let $\mathcal{L}$ be the remaining nodes, $d_{N'+1}$ through $d_N$. The algorithm consists of two steps, corresponding to the packing of nodes 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 $l_i = \langle l_i^1, \ldots, l_i^{\Delta}\rangle$ where $l_i^j$ denotes the number of unpopular size $a_j$ data items whose clients are assigned (completely) to a node $d_i \in \mathcal{H}$.

Since all nodes are identical, we can guess each such vector in $O(N^{k+1^{\Delta}})$ 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 $(k+1)^{\Delta}$, 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 $\sum_j l_i^j a_j \leq k$.) 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 nodes 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 node $d_i$ the largest size $l_i^j$ size $a_j$ unpopular data items that remain.

Next we develop a dynamic program moving across the nodes from 1 through $N'$ so as to find an optimal $(\epsilon'^2/k)$-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 node is restricted to be $(1-\epsilon')L$ and the number of data items allowed in node $d_i$ 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 $\beta$-tuple $\vec{v} = \langle v_1^1, v_2^1, ..., v_{\beta}^1, v_1^2, v_2^2 \ldots, v_{\beta}^2, \ldots, v_1^{\Delta}, v_2^{\Delta}, \ldots, v_{\beta}^{\Delta}\rangle$ 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 nodes $d_1$

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 $(Nk)^{\Delta k/\epsilon'^3}$, which is polynomial for any fixed $\epsilon'$.

Packing items in $\mathcal{L}$: We know that our solution need not assign clients corresponding to a popular data item to more than one node in $\mathcal{L}$. Moreover, at most $\epsilon' L$ clients from any popular data item are packed in a node 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 $\Delta$ lists of items, $L'_i$ $(i = 1 \ldots \Delta)$where $L'_i$ has both popular and unpopular items of size $a_i$. The popular items are truncated as mentioned above.

We have exactly $N - N'$ nodes that are light nodes, and we wish to obtain an optimal packing of these light nodes using the $\Delta$ 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 node. 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 $i = 1 \ldots \Delta$ we guess $n_i$, the number of data items from $L'_i$ that are chosen to be packed in light nodes. Since there are $O(M^{\Delta})$ such choices, this is a polynomially bounded search space. For each such choice, 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 nodes. 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 node by a vector $(x_1, x_2, \ldots x_{\Delta})$ where $x_i$ is the number of items of size $a_i$ packed in this node. For this to be feasible, it must satisfy the property that $\sum_{i=1}^{\Delta} a_i x_i \leq k$. 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 $O(k^{\Delta})$, in other words a constant for fixed $k$ and $\Delta$. Hence we obtain the fact that each light node is characterized by a constant number of (feasible) types

$T^{(1)}, \ldots, T^{(\alpha)}$ where $T^{(j)} = (x_1^j, \ldots, x_\Delta^j)$.

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

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

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

The first constraint simply specifies that the total number of nodes of each type is exactly the total number of light nodes. 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 [113] 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.

# Chapter 3

# Centralized Data Reconfiguration

This is joint work with Samir Khuller, Yung-Chun (Justin) Wan and Leana Golubchik. These results also appeared in [106].

## 3.1 Introduction

We live in an era of data explosion and this data explosion necessitates the use of large storage systems. Storage Area Networks (or SANs) are the leading [170] infrastructure for enterprise storage. A SAN essentially allows multiple processors to access several storage devices. They typically access the storage medium as though it were one large shared repository. One crucial function of such a storage system is that of deciding the placement of data within the system. This data placement is dependent on the demand pattern for the data. For instance, if a particular data item is very popular the storage system might want to host it on a node with high bandwidth or make multiple copies of the item. The storage system needs to be capable of handling flash crowds [98]. During events triggered by such flash crowds, the demand distribution becomes highly skewed and different from the normal demand distribution.

It is known that the problem of computing an optimal data placement (An optimal placement will allow a maximum number of users to access information of their interest) for a given demand pattern is NP-Hard [70]. However, polynomial time approximation schemes as well as efficient combinatorial algorithms that compute almost optimal solutions are known for this problem [156, 155, 70]. So we can assume that a near-optimal placement can be computed once a demand pattern is specified.

As the demand pattern changes over time and the popularity of items changes, the storage system will have to modify its internal placement accordingly. Such a modification in placement will typically involve movement of data items from one set of nodes to another or requires changing the number of copies of a data item in the system. For such a modification to be effective it should be computed and applied quickly. In this work we are concerned with the problem of finding such a modification i.e., modifying the existing placement to efficiently deal with a new demand pattern for the data. This problem is referred to as the data migration problem and was considered in [109, 71]. The authors used a data placement algorithm to compute a new "target" layout. The goal was to "convert" the existing layout to the target layout as quickly as possible. The communication model that was assumed was a half-duplex model where a matching on the nodes can be fixed, and for each matched pair one can transfer a single object in a round. The goal was to minimize the number of rounds taken. The paper developed constant factor approximation algorithms for this NP-hard problem [109]. In practice these algorithms find solutions that are reasonably close to optimal. However, even when there is *no* drastic change in the demand distribution it can still take many rounds of migration to achieve the new target layout. This happens since the scheme completely disregards the existing placement in trying to compute the target placement.

In this chapter we consider a new approach to dealing with the problem of changes in the demand pattern. We ask the following question:

*In a given number of migration rounds, can we obtain a layout by making changes to the existing layout so that the resulting layout will be the best possible layout that we can obtain within the specified number of rounds?*

Of course, such a layout is interesting only if it is significantly better than the existing layout for the new demand pattern.

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

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

Our method can therefore be used to *quickly* transform an existing placement to deal with changes in the demand pattern. We *do not* make any assumptions about the type of demand changes – hence the method can be used to quickly deal with any type of change in the demands. We also show that the problem of finding an optimal set of changes that can be applied in one round is NP-hard (see Section 3.2.3 for the proof). The proof demonstrates that some unexpected data movement patterns can yield a high benefit.

In the remaining part of the introduction, we present the model and the assumptions made, and restate our result formally.

### 3.1.1 Model summary

We consider the following model for our storage system. There are $N$ parallel nodes that form a *Storage Area Network*. Each node has a storage capacity of $K$ and has a load handling capacity (or bandwidth) of $L$. Each node can be viewed as a collection of physical disks - i.e. as a ”logical disk”.

The efficiency of the system depends crucially on the *data layout* pattern that is chosen for the nodes. This data layout pattern or data placement specifies for each item, which set of nodes it is stored on (note that the whole item is stored on each of the nodes specified by the placement, so these are copies of the item). The next problem is that of mapping the demand for data to nodes. Each node has an upper bound on the total demand that can be mapped to that node. A simple way to find an optimal assignment of demand to nodes, is by running a single network flow computation in an appropriately defined graph (see Section 3.2.1).

Different communication models can be considered based on how the nodes are connected. We use the same model as in [12, 82] where the nodes may communicate on any matching; in other words, the underlying communication graph allows for communication between any pair of devices via a matching (e.g., as in a switched storage network with unbounded backplane bandwidth). *This model best captures an architecture of parallel storage devices that are connected on a switched* *network with sufficient bandwidth. This is most appropriate for our application.* This model is one of the most widely used in all the work related to gossiping and broadcasting. These algorithms can also be extended to models where the size of the matching in each round is constrained [109]. This can be done by a simple simulation, where we only choose a maximal subset of transfers to perform in each round.

Suppose we are given an initial demand pattern $\mathcal{I}$. We use this to create an initial layout $L_{\mathcal{I}}$. Over time, the demand pattern for the data may change. At some point of time the initial layout $L_{\mathcal{I}}$ may not be very efficient. At this point the storage manager may wish to re-compute a new layout pattern. Suppose the target demand pattern is determined to be $\mathcal{T}$ (this could be determined based on the recent demand for data, or based upon projections determined by previous historical trends). Our goal is to migrate data from the current layout to a new layout. We would like this migration to complete quickly since the system is running inefficiently in addition to using a part of its local bandwidth for migrating data. It is therefore desirable to complete the conversion of one layout to another layout quickly. However, note that previous methods completely ignored the current layout and fixed a target layout $L_{\mathcal{T}}$ based on the demand $\mathcal{T}$. *Is it possible that there are layouts $\mathcal{L}'$ with the property that they are almost as good as $L_{\mathcal{T}}$, however, at the same time we can "convert" the initial layout $L_{\mathcal{I}}$ to $\mathcal{L}'$ in very few rounds (say compared to the number of rounds required to convert $L_{\mathcal{I}}$ to $L_{\mathcal{T}}$)?* It is our objective to consider this central question in this chapter. In fact, we answer the question in the affirmative by doing a large set of experiments.

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

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

Of course there is no reason to assume that repeatedly solving the one round problem will actually yield an optimal solution for the $\ell$ round version of this problem. However, as we will see, this approach is very effective.

## 3.2 The problem

### 3.2.1 Example

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

In this toy example, we consider a storage system that consists of 4 identical nodes. Each node has storage capacity of 3 units and load capacity (or bandwidth) of 100 units. There are 9 data items that need to be stored in the system. The initial demand distribution $\mathcal{I}$ and the new demand distribution $\mathcal{T}$ are as follows:

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

The placement $L_{\mathcal{I}}$ (which in this case is also an optimal placement) obtained using the sliding window algorithm. The sliding window algorithm proposed by Shachnai and Tamir [156] is currently the best practical algorithm for this problem. For more on the sliding window algorithm and its performance, see [70]. for the demand distribution above is as follows (the numbers next to the items on nodes indicates the mapping of demand to that copy of the item):

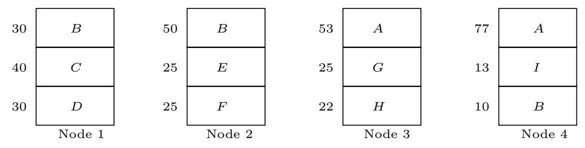


**Figure 3.1:** Optimal placement $L_{\mathcal{I}}$ for the initial demand distribution $\mathcal{I}$, satisfies all the demand. Storage capacity $K$=3, Bandwidth L=100. In addition to producing the layout the sliding window algorithm finds a mapping of demand to nodes, which is optimal for the layout computed.

To determine the maximum amount of demand that the current placement $L_{\mathcal{I}}$ can satisfy for the new demand distribution $\mathcal{T}$, we compute the max-flow in a network constructed as follows. In this network we have a node corresponding to each item and a node corresponding to each node. We also have a source and a sink vertex. We have edges from item vertices to node vertices if in the placement $L_{\mathcal{I}}$, that item was put on the corresponding node. Capacities of edges from the source to every item is equal to the demand for that item in the new distribution. The rest of the edges have capacity equal to the node bandwidth. Using the flow network above, we can re-assign the demand $\mathcal{T}$ using the same placement $L_{\mathcal{I}}$ as given in Figure 3.3. Figure 3.2 shows the flow network obtained by applying the construction described above, corresponding to the initial placement $L_{\mathcal{I}}$ and new demand $\mathcal{T}$.

A small change can convert $L_{\mathcal{I}}$ to an optimal placement. In general, we would like to find changes that can be applied to the existing placement in a single round and get a placement that is close to an optimal placement for the new demand distribution. In a round a node can either be the source or the target of a data transfer but not both. In fact, in this example a single change that involves copying an item from one node to another is sufficient (and does not involve the other two nodes in data transfers). This is illustrated in Figure 3.4.

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

We compare this approach to that of previous works [109, 71] which completely disregard

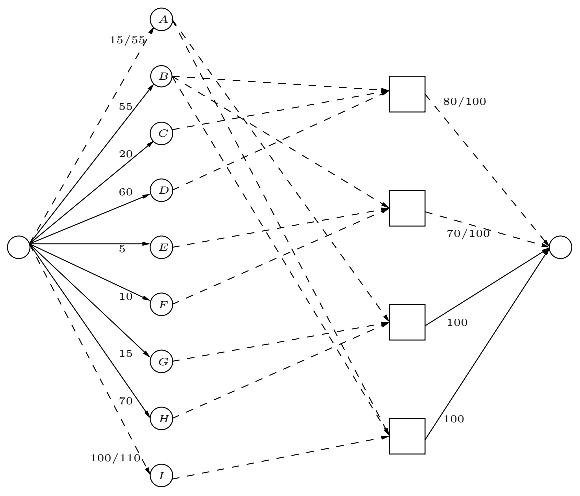


**Figure 3.2:** Flow network to determine maximum benefit of using placement $L_\mathcal{I}$ with demand distribution $\mathcal{T}$. $L_\mathcal{I}$ is sub-optimal for $\mathcal{T}$ and can only satisfy 350 out of a maximum of 400 units of demand. Saturated edges are show using solid lines.

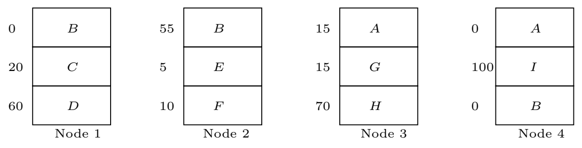


**Figure 3.3:** Maximum demand that placement $L_\mathcal{I}$ can satisfy for the new demand distribution $\mathcal{T}$. $L_\mathcal{I}$ is sub-optimal for $\mathcal{T}$ and can only satisfy 350 out of a maximum of 400 units of demand.

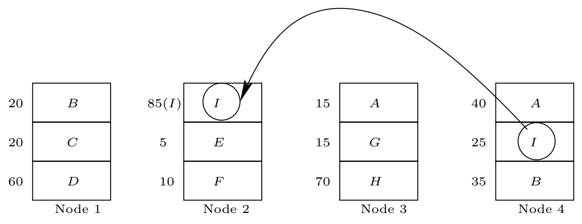


**Figure 3.4:** Removing item $B$ from node 2 and replacing it with a copy of item $I$ from node 4 converts $L_\mathcal{I}$ to an optimal placement $\mathcal{L}'$ for the new demand distribution $\mathcal{T}$. The placement shown above is optimal for $\mathcal{T}$ and satisfies all demand.

the existing placement and simply try to minimize the number of parallel rounds needed to convert the existing placement to an optimal placement for the new demand distribution. In Fig. 3.6, we show that using the old approach, it takes 4 rounds of transfers to achieve what our approach did in a single round (and using just one transfer). In Figure 3.5 an optimal placement $L_\mathcal{T}$ is recomputed, using the sliding window algorithm for computing a placement for a given demand, for the new demand distribution $\mathcal{T}$. We show in Figure 3.6 the smallest set of transfers required to convert $L_{\mathcal{I}}$ to $L_{\mathcal{T}}$. *Note that both placement $\mathcal{L}'$ (obtained after the transfer shown in Figure 3.4 is applied) and placement $L_{\mathcal{T}}$ shown in Figure 3.5 are optimal placements for the new demand distribution $\mathcal{T}$.* Note that this is an optimal solution that also addresses the space constraint on the node (this property is not actually maintained by the data migration algorithms developed earlier [109]).

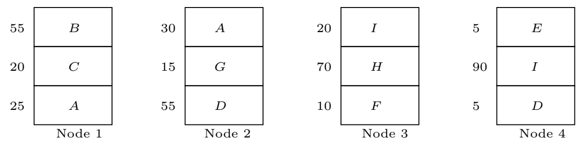


**Figure 3.5:** Placement $L_{\mathcal{T}}$. Output of the Sliding window algorithm for the new demand distribution $\mathcal{T}$.

### 3.2.2 Formal definition

The storage system consists of $N$ nodes. Each node has load-capacity of $L$ and a storage-capacity of $K$. We have $m$ items, each item $j$ has size 1 and demand $\ell_j$. This constitutes an $m$-dimensional demand distribution $\ell = (\ell_1, \ldots, \ell_m)$. An $m$-dimensional *placement* vector $p_i$ for a node $i$ is $(p_{i1}, \ldots, p_{im})$ where $p_{ij}$ are $0-1$ entries indicating that item $j$ is on node $i$. An $m$-dimensional *demand* vector $d_i$ for a node $i$ is $(d_{i1}, \ldots, d_{im})$ where $d_{ij}$ is the demand for item $j$ assigned to node $i$. Define $\mathcal{V}(\{d_i\}) = \sum_i \sum_j d_{ij}$ as the benefit of the set of demand vectors $\{d_i\}$. A set of placement and demand vectors that satisfy the following constraints is said to constitute a feasible placement and demand assignment:

1. $\sum_j p_{ij} \leq K$ for all nodes $i$. This ensures that the storage-capacity is not violated.

2. $\sum_j d_{ij} \leq L$ for all nodes $i$. This ensures that the load-capacity is not violated.

3. $d_{ij} \leq p_{ij}\ell_j$. This ensures that the demand for an item $j$ is routed to node $i$ only if that item is present on node $i$.

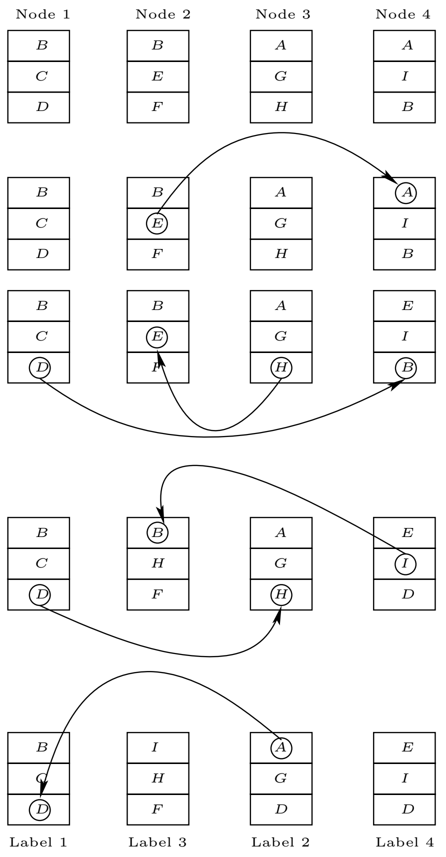


**Figure 3.6:** Transforming $L_{\mathcal{I}}$ to $L_{\mathcal{T}}$ takes 4 rounds. Note that the nodes here will need to be renumbered to match the sliding window output. Final node 2 corresponds to node 3 in the sliding window output, final node 3 corresponds to node 2 in the sliding window output.

4. $\sum_i d_{ij} \leq \ell_j$ for all items $j$. This ensures that no more than the total demand for an item is packed.

A one-round-migration is essentially a matching on the set of nodes. More formally, a one-round-migration is a 0-1 function $\Delta(s_d, s_i, t_d, t_i)$ where $s_d, t_d \in \{1, \dots, N\}$ and $s_i, t_i \in \{1, \dots, m\}$. Here $s_d$ is the source node, $s_i$ is the source item, $t_d$ is the target node, $t_i$ is the target item. Further, $\Delta(.)$ has to satisfy the following conditions:

1. $\sum_{t_d} \sum_{s_i} \sum_{t_i} \Delta(s_d, s_i, t_d, t_i) \leq 1$ for all nodes $s_d$. This ensures that a node can be the source for at most one transfer.

2. $\sum_{s_d} \sum_{s_i} \sum_{t_i} \Delta(s_d, s_i, t_d, t_i) \leq 1$ for all nodes $t_d$. This ensures that a node can be the target for at most one transfer.

3. $\left(\sum_{s_d} \sum_{s_i} \sum_{t_i} \Delta(s_d, s_i, t_d, t_i)\right) + \left(\sum_{t_d} \sum_{s_i} \sum_{t_i} \Delta(s_d, s_i, t_d, t_i)\right) \leq 1$ for all node pairs $s_d = t_d$. This ensures that a node can simultaneously not be both a source and a target.

4. $(s_d = t_d) \Rightarrow \Delta(s_d, *, t_d, *) = 0$. This ensures that there are no self loops in the transfer graph.

5. $\sum_{t_d} \sum_{t_i} \Delta(s_d, s_i, t_d, t_i) \leq p_{s_d s_i}$ for all nodes $s_d$ and items $s_i$. This ensures that a node $s_d$ can be source of an item $s_i$ only if that item is on that node (i.e. $p_{s_d s_i} = 1$).

We can apply this function to an existing placement to obtain a new placement as follows. If $\Delta(s_d, s_i, t_d, t_i) = 1$, then set $p_{t_d t_i} = 0$ and $p_{t_d s_i} = 1$. We compute the optimal demand assignment for the new placement using max-flow.

ONE-ROUND-MIGRATION: When given an initial demand distribution $\ell_{initial}$, a corresponding set of feasible placement vectors $\{p_i\}$, and demand vectors $\{d_i\}$ and a final demand distribution $\ell_{final}$, the problem asks for a one-round-migration $\Delta(.)$ that when applied to the initial placement yields placement vectors $\{p_i^*\}$ and demand vectors $\{d_i^*\}$ such that $\mathcal{V}(\{d_i^*\})$ is maximized.

We show that this problem is NP-Hard (See Section 3.2.3 for a proof).

### 3.2.3 Hardness proof

Recall that the *Subset-Sum* Problem is known to be $NP$-complete [64]. The *Subset-Sum* problem is defined as follows: Given a set $S = \{a_1, \ldots, a_n\}$ and a number $b$, where $a_i, b \in \mathcal{Z}^+$. Does there exist a subset $S' \subset S$ such that $\sum_{a_j \in S'} a_j = b$? Let $\text{sum}(S) = \sum_{a_i \in S} a_i$.

The *One-Round Migration* problem is defined as follows. We are given a collection of identical nodes $D_1, \ldots D_N$. Each node has a storage capacity of $K$, and a load capacity of $L$. We are also given a collection of data objects $M_1, \ldots M_M$, and a layout of the data objects on the nodes. The layout specifies the subset of $K$ data objects stored on each node. Each data object $M_i$ has demand $u_i$. The demand for any data object may be assigned to the set of nodes containing that object (demand is splittable), without violating the load capacity of the nodes. For a given layout, there may be no solution that satisfies all the demand. Is there a one-round migration to compute a new layout in which all the demand can be satisfied?

A one-round migration is a matching among the nodes, such that for each edge in the matching, one source node may send an item to a node that it is matched to (half-duplex model).

We show that the One-Round Migration problem is $NP$-hard by reducing Subset-Sum to it. We will create a set of $N = 3n + 4$ nodes, each having capacity two ($K = 2$). There are $4n + 6$ items in all. We will assume that $L$ is very large. The current layout is shown in Figure 3.7.

The demand for various items is as follows: Demand for $G_i$ is $L - a_i$. Demand for $C_i = \frac{L}{2} + a_i$. Demand for $E_i = \frac{L}{2}$. Demand for $F_i = L - a_i$.

Demand for $A = \text{sum}(A) + \frac{L}{2}$. Demand for $H = \text{sum}(A) + \frac{L}{2}$. Demand for $X = \frac{L}{2}$. Demand for $Y = L - b$. Demand for $Z = L - (\text{sum}(A) - b)$. Demand for $W = \frac{L}{2}$.

If we assume that the demand for $C_i$ is $\frac{L}{2}$ then the assignment shown can satisfy all the demand. We will assume that all but two of the nodes are load saturated (total assigned demand is exactly $L$). *If the demand for $C_i$ increases by $a_i$*, then we have to re-assign some of this demand. The claim is that all of the demand can be handled after one round of migration if and only if there is a solution to the subset-sum instance. It is clear that a given solution (a matching) can be verified in polynomial time.

($\Rightarrow$) Suppose there is a subset $S' \subset S$ that adds exactly to $b$. We copy $H$ (from the node containing $H$ and $W$) to the node containing $Z$, and $A$ (from the node containing $X$ and $A$) to the node containing $Y$. If $a_i \in S'$ then we copy $C_i$ to the node containing $G_i$, and over-write the copy of $A$ on that node. All clients for $A$ from this set of nodes can be moved to the node containing $A$ and $Y$. If $a_i \notin S'$ then we copy $C_i$ to the node containing $F_i$ and over-write the copy of $H$ on that node. All clients for $H$ from this set of nodes can be moved to the node containing $H$ and $Z$.

($\Leftarrow$) First note that the total demand is $3nL + 4L$. Since there are $3n + 4$ nodes, all nodes must be load saturated for a solution to exist. We leave it for the reader to verify that with the current layout there is no solution that meets all the demand. Suppose there exists a one-round migration that enables a solution where all of the demand can be assigned. A new copy has to be

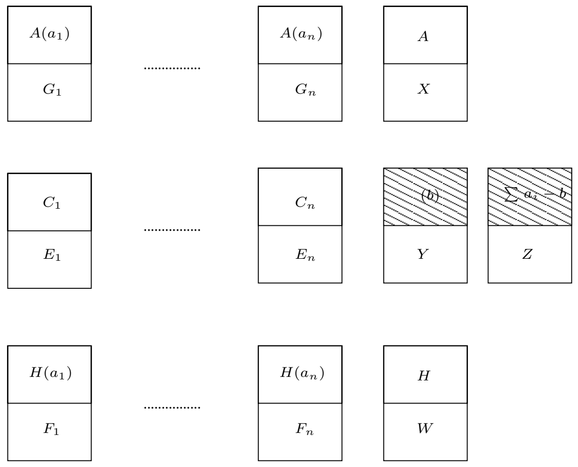


**Figure 3.7:** Reduction from SUBSET-SUM to ONE-ROUND-MIGRATION. Shaded portion indicates empty space. Number within brackets following item name indicates the amount of load assigned to the item.

created for each $C_i$, or $E_i$ since the total load for $C_i$ and $E_i$ is $L + a_i$, and exceeds $L$. Assume w.lo.g that a copy of $C_i$ will be made to handle the excess demand of $a_i$ on this node. We also assume without loss of generality that $a_i < b$ so moving $C_i$ to one of the nodes containing $Y$ or $Z$ would not be of much use in load saturating those nodes. The only choice is to decide whether this new copy is made at the expense of a copy of $H$ or at the expense of a copy of $A$. Note that $C_i$ cannot overwrite any of the other items since only a single copy of these items exists in the system. Since this is a one-round migration, we cannot move a single copy of an item to another node, and then re-write it subsequently. Note that $C_i$ has to overwrite the corresponding $A$ node or $H$ node, otherwise we will be unable to recover all the demand. Since the nodes containing $Y$ and $Z$ are also load saturated, we will copy an item onto those nodes. Moreover we have to move one item (either $A$ or $H$) to the node containing $Y$. Suppose that $A$ is copied to the node containing $Y$ and $H$ is copied to the node containing $Z$. (The reverse case is similar.) When we shift $b$ amount of demand of $A$ to the node containing $Y$, we have to completely remove the demand from a node containing $A$, otherwise we will lose some demand. If $C_i$ is moved to a node containing $A$ then $a_i \in S'$. If $C_i$ is moved to a node containing $H$ then $a_i \notin S'$. Since $C_i$ over-writes $A$ ($H$), all of the demand of $A$ ($H$) is moved out of the node. Clearly, the total size of $S'$ must be exactly $b$.

## 3.3 Algorithm for one round migration

For any node $d$, let $I(d)$ denote the items on that node.

Corresponding to any placement $\{\mathbf{p_i}\}$ (a placement specifies for each item, which set of nodes it is stored on), we define the corresponding flow graph $G_p(V, E)$ as follows. We add one node $a_i$ to the graph for each item $i \in \{1 \dots m\}$. We add one node $d_j$ for each node $j \in \{1 \dots N\}$. We add one source vertex $s$ and one sink vertex $t$. We add edges $(s, a_i)$ for each item $i$. Each of these edges have capacity $\texttt{demand}(i)$ (where $\texttt{demand}(i)$ is the demand for item $i$). We also add edges $(d_j, t)$ for each node $j$. These edges have capacity $L$ (where $L$ is the load capacity of node $j$). For every node $j$ and for every item $i \in I(j)$, we add an edge $(a_i, d_j)$ with capacity $L$.

The algorithm starts with the initial placement and works in phases. At the end of each phase, it outputs a pair of nodes and a transfer corresponding to that node pair.

We determine the node and transfer pair as follows. Consider a phase $r$. Let $\{\mathbf{p_i}\}_r$ be the current placement. For every pair of nodes $d_i$ and $d_j$, for every pair of items $(a_i, a_j)$ in $I(d_i) \times I(d_j)$, modify the placement $\{\mathbf{p_i}\}_r$ to obtain $\{\mathbf{p_i^{'}}\}_r$ by overwriting $a_j$ on $d_j$ with $a_i$. Compute the max-flow in the flow graph for the placement $\{\mathbf{p_i^{'}}\}_r$. Note down the max-flow value and revert the placement back to $\{\mathbf{p_i}\}_r$. After we go through all pairs, pick the $(a_i, a_j)$ transfer pair and the corresponding $(d_i, d_j)$ node pair that resulted in the flow-graph with the largest max-flow value. Apply the transfer $(a_i, a_j)$ modifying placement $\{\mathbf{p_i}\}_r$ to obtain $\{\mathbf{p_i}\}_{r+1}$ - which will be the starting placement for the next phase. We can no longer use nodes $d_i$ and $d_j$ in the next phase. Repeat until there is no pair that can increase the max-flow or till we run out of nodes.

## 3.4 Speeding up the algorithm

The algorithm described in Section 3.3 recomputes max-flow in the flow graph from scratch when evaluating each move. Recall that the algorithm proceeds in phases and at the end of each phase, it identifies a pair of nodes $(d_i, d_j)$ and a $(a_i \in I(d_i), a_j \in I(d_j))$ transfer for that pair of nodes.

We can speed up the algorithm by observing that the max-flow value increases monotonically from one phase to the next and therefore we need not recompute max-flow from scratch for each phase. Rather, we compute the residual network for the flow graph once and then make incremental changes to this residual network for each max-flow computation. All max-flow computations in this version of the algorithm are computed using the Edmonds-Karp algorithm (see [8]). Let $G_i$ denote the residual graph at the end of phase $i$. Let $G_0$ be the residual graph corresponding to the initial graph. All max-flow computations in phase $i+1$, we begin with the residual graph $G_i$ and find augmenting paths (using BFS on the residual graph) to evaluate the max-flow. After each transfer pair in phase $i+1$ is considered, we undo the changes to the residual graph and revert back to $G_i$. At the end of phase $i+1$, we apply the best transfer found in that phase, recompute max-flow and use the corresponding residual graph as $G_{i+1}$.

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

Variant 1: For every pair of nodes $d_i$ and $d_j$, let $I_+(d_i)$ be the set of items on node $d_i$ that have **unsatisfied demand**. For every pair of items $(a_i, a_j)$ in $I_+(d_i) \times I(d_j)$, overwrite $a_j$ on $d_j$ with $a_i$, compute the max-flow. Pick the $(a_i, a_j)$ pair that gives the largest increase in the max-flow value. Repeat till there is no pair that can increase the max-flow or until we run out of nodes.

Variant 2: For every pair of nodes $d_i$ and $d_j$, let $I_+(d_i)$ be the set of items on node $d_i$ that have **unsatisfied demand** and $I_-(d_j)$ be the items with **lowest demand** on node $d_j$. For every pair of items $(a_i, a_j)$ in $I_+(d_i) \times I_-(d_j)$, overwrite $a_j$ on $d_j$ with $a_i$, compute the max-flow. Pick the $(a_i, a_j)$ pair that gave the largest increase in the max-flow value. Repeat till there is no pair that can increase the max-flow or until we run out of nodes.

All the experimental results that we present in Section 3.5 are obtained using the second variant (described above). Experiments were run on a 2.8Ghz Pentium 4C processor with 1GB

RAM running Ubuntu Linux 5.04. To solve even the largest instances in our experiments, a C (gcc 3.3) implementation of the second variant took only a couple of seconds while the brute force algorithm took on the order of several hours.

## 3.5 Experiments

In this section, we describe the experiments used to evaluate the performance of our heuristic and compare it to the old approach to data migration. The framework of our experiments is as follows:

1. (*Create an initial layout*) Run the sliding window algorithm [70], given the number of user requests for each data object.

2. (*Create a target layout*) To obtain a target layout, we take one of the following approaches.

   (a) Shuffle method 1: Initial demand distribution is chosen with Zipf (will be defined later in this section) parameter 0.0 (high-skew). To generate the target distribution, pick 20% of the items and promote them to become more popular items.

   (b) Shuffle method 2: Initial demand distribution is chosen with Zipf parameter 0.0 (high-skew). To generate the target distribution, the lowest popularity item is promoted to become the most popular item.

   (c) Shuffle method 3: The initial demand distribution is chosen with Zipf parameter 1.0 (uniform-distribution). The target distribution is chosen with Zipf parameter 0.0 (high-skew).

   (d) Shuffle method 4: The initial demand distribution is chosen with Zipf parameter 0.0 (high-skew). The target distribution is chosen with Zipf parameter 1.0 (uniform-distribution).

3. Record the number of rounds required by the old data migration scheme to migrate the initial layout to the target layout.

4. Record the layout obtained in each round of our heuristic. Run 10 successive rounds of our one round migration starting from the initial layout. The layout output after running these 10 successive rounds of our heuristic will be considered as the final layout output by our heuristic.

We note that few large-scale measurement studies exist for the applications of interest here (e.g., video-on-demand systems), and hence below we are considering several potentially interesting distributions. Some of these correspond to existing measurement studies (as noted below) and others we consider in order to explore the performance characteristics of our algorithms and to further improve the understanding of such algorithms. For instance, a Zipf distribution is often used for characterizing people's preferences.

*Zipf Distribution* The Zipf distribution is defined as follows:

$$\text{Prob(request for item } i) = \frac{c}{i^{1-\theta}} \qquad \begin{array}{c} \forall i = 1, \ldots, M \\ \text{and} \\ 0 \leq \theta \leq 1 \end{array}$$

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

and $\theta$ determines the degree of skewness. For instance, $\theta = 1.0$ corresponds to the uniform distribution, whereas $\theta = 0.0$ corresponds to the skewness in access patterns often attributed to movies-on-demand type applications. See for instance the measurements performed in [32]. Flash crowds are also known to skew access patterns according to Zipf distribution [98]. In the experiments below, Zipf parameters are chosen according to the shuffle methods described earlier in the section.

We now describe the storage system parameters used in the experiments, namely the number of nodes, space capacity, and load capacity (the maximum number of simultaneous user requests that a node may serve).

In the first set of experiments, we used a value of 60 nodes. We tried three different pairs of settings for space and load capacities, namely: (A) 15 and 40, (B) 30 and 35, and (C) 60 and 150.

In the second set of experiments, we varied the number of nodes from 10 to 100 in steps of 10. We used a value of K=60, L=150 (this is the 3rd pair of L,K values used in the first set of experiments).

We obtained these numbers from the specifications of modern SCSI hard drives. For example, a 72GB 15,000 rpm node can support a sustained transfer rate of 75MB/s with an average seek time of around 3.5ms. Considering MPEG-2 movies of 2 hours each with encoding rates of 6Mbps, and assuming the transfer rate under parallel load is 40% of the sustained rate, the node can store 15 movies and support 40 streams. The space capacity 30 and the load capacity 35 are obtained from using a 150GB 10,000 rpm node with a 72MB/s sustained transfer rate. The space capacity 60 and the load capacity 150 are obtained by assuming that movies are encoded using MPEG-4 format (instead of MPEG-2). So a node is capable of storing more movies and supporting more streams. For each tuple of N,L,K and shuffle method we generated 10 instances. These instances were then solved using both our heuristic as well as the old data migration heuristic. The results for each N,L,K and shuffle method tuple were averaged out over these 10 runs.

### 3.5.1 Results and Discussion

Figures 3.8a, 3.9a, 3.9b, 3.9c and Tables 3.1a, 3.1b, 3.1c correspond to the first set of experiments. Figures 3.10a, 3.10b, 3.10c, 3.10d, 3.8b and Tables 3.2a, 3.2b, 3.2c, 3.2d correspond to the second set of experiments.

Figures 3.9a, 3.9b, 3.9c, 3.10a, 3.10b, 3.10c, 3.10d compare the solution quality of our heuristic with that of the old approach. Tables 3.1a, 3.1b, 3.1c, 3.2a, 3.2b, 3.2c, 3.2d and Figure 3.8a compares the number of rounds taken by our approach with the number of rounds taken by the old approach to achieve similar solution quality.

We highlight the following observations supported by our experimental results:

- In all our experiments, our heuristic was able to get within 8% of the optimal solution using 10 rounds. This can be seen in all the figures and tables. For instance, see Figure 3.8a.
- In comparison (see Figure 3.8a and Tables 3.1a, 3.1b, 3.1c, 3.2a, 3.2b, 3.2c, 3.2d), the old scheme took a significantly larger number of rounds. For example, in the case of K=60, L=150 (corresponding to storing video as mpeg-4) the old scheme took over 100 rounds for

every shuffle method and for every value of N we used, while our scheme was able to achieve similar solution quality within 10 rounds.

- Response to change in demand distribution: The experiments reveal an interesting behavior of the heuristic. When the target demand distribution is highly skewed, the heuristic's response or the amount of improvement made in successive rounds is linear. In contrast, when the demand is less skewed (i.e. the demand distribution is significantly different from the initial distribution but still the target distribution is not very skewed), the response is much sharper. For example in Figure 3.8b, consider the response curve for shuffle methods 4 and 2 (low-skew) and contrast it with the flat response curves for shuffle methods 1 and 3 (high-skew).
  - Sharp response or diminishing returns: For a concrete example; in Figure 3.10a the improvement obtained by our heuristic in the first round is almost as high as 10%, but successive improvements taper off quickly. This probably happens because we use a greedy algorithm and most of the gains are made in the first round and since this type of behavior is observed mainly when the demand is less skewed, there are presumably several items that need to replicated.
  - Flat response: For a concrete example; in Figure 3.10c the improvement obtained by our heuristic for N=100 in the first two rounds (1 and 2) is just about twice the benefit obtained in the last two rounds (9 and 10). This is probably because most of the load is concentrated on a few items and there is a large amount of unsatisfied demand. In each round we make more copies of these high popularity items and see almost the same benefit in each round.
- The case for this type of approach (that of making small changes to existing placement in consecutive rounds) is best supported by results from Table 3.2d. This is an example of a case where the existing placement is already very good for the target distribution. The storage manager may wish to do a few rounds of migration to recover the amount of lost

load. Our scheme lets the storage manager do such a quick adaptation. In contrast the old scheme takes over 150 rounds on average to achieve comparable results. This is especially unacceptable given that we already start off with a pretty good placement. In fact, shuffle method 4 seemed to consistently trigger expensive migrations in the old scheme while our scheme was able to get close to optimal within a couple of rounds. This is not surprising since the old scheme completely disregards the existing placement.

- Shuffle method 3 seemed to produce "harder" instances for our heuristic compared to the other shuffle methods we tried. This is not surprising since shuffle method 3 makes a drastic change to the demand distribution (moving it from uniform to highly skewed Zipf).

- It is very promising that our scheme performs particularly well for shuffle methods 1 and 2 (which is the type of demand change we expect to see in practice).

| Shuffle method | Our Scheme | | Old Scheme | |
|---|---|---|---|---|
| | Rounds | Demand % | Rounds (avg) | Demand % |
| 1 | **10** | 99.10 | **41.8** | 99.92 |
| 2 | **10** | 98.86 | **39.1** | 99.92 |
| 3 | **10** | 97.26 | **43.7** | 98.91 |
| 4 | **10** | 99.04 | **54.2** | 100 |

**(a)** Comparison of old scheme with our scheme for N=60, K=15, L=40

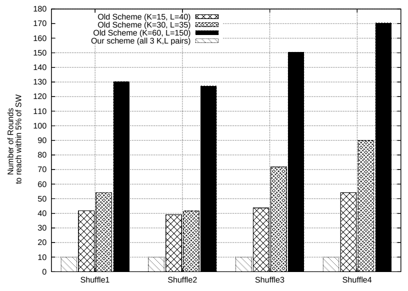
| Shuffle method | Our Scheme | | Old Scheme | |
|---|---|---|---|---|
| | Rounds | Demand % | Rounds (avg) | Demand % |
| 1 | **10** | 98.50 | **54.2** | 99.83 |
| 2 | **10** | 97.72 | **41.6** | 99.79 |
| 3 | **10** | 97.02 | **71.8** | 98.99 |
| 4 | **10** | 98.57 | **89.9** | 100 |

**(b)** Comparison of old scheme with our scheme for N=60, K=30, L=35

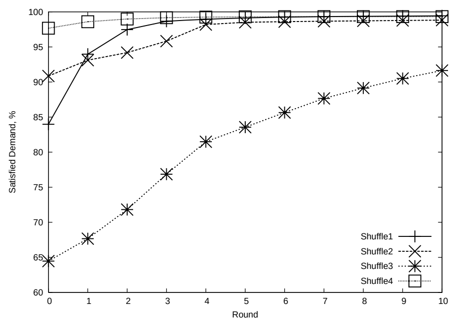
| Shuffle method | Our Scheme | | Old Scheme | |
|---|---|---|---|---|
| | Rounds | Demand % | Rounds (avg) | Demand % |
| 1 | **10** | 99.41 | **130.3** | 99.98 |
| 2 | **10** | 98.54 | **127.3** | 99.99 |
| 3 | **10** | 94.41 | **150.4** | 99.68 |
| 4 | **10** | 99.30 | **170.5** | 100 |

**(c)** Comparison of old scheme with our scheme for N=60, K=60, L=150

**Table 3.1:** Experimental Results



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

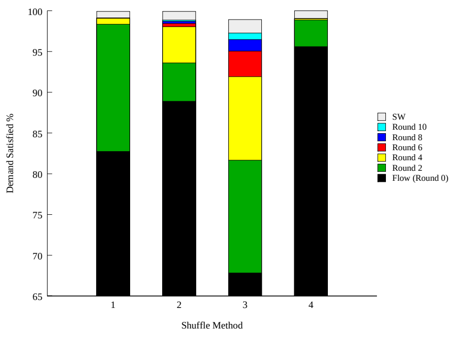


**(b)** Plot comparing the response of our heuristic to the various shuffle methods. The response to shuffle 3 and shuffle 2 is much flatter than the diminishing returns type of response for shuffle 4 and shuffle 1. We used N=100, K=60, L=150 for each experiment. Every data point was obtained by averaging over 10 runs.

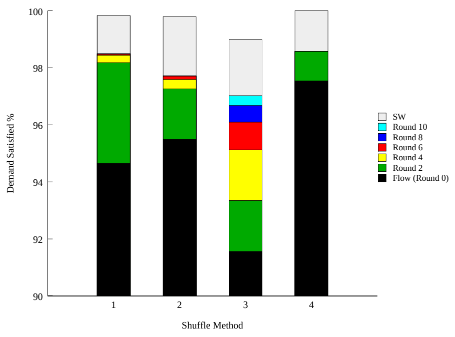
**Figure 3.8:** Experimental Results

## 3.6 Conclusion

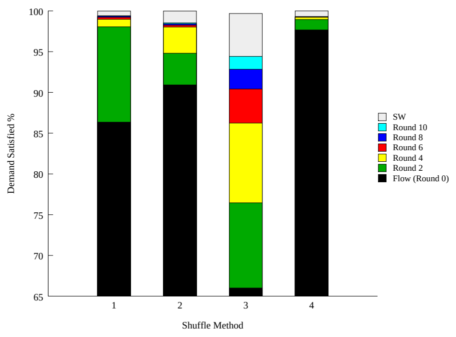
We proposed a new approach to deal with the problem of changing demand. We defined the one-round-migration problem to aid us in our effort. We showed that the one-round-migration problem is NP-Hard and that unexpected data movement patterns can yield high benefit. We gave heuristics for the problem. We gave experimental evidence to suggest that our approach of doing a few rounds of one-round-migration consecutively performs very well in practice. In particular, in all our experiments they were able to quickly adapt the existing placement to one that is close to the optimal solution for the changed demand pattern. We showed that, in contrast, previous approaches took many more rounds to achieve similar solution quality.



**(a)** N=60, K=15, L=40

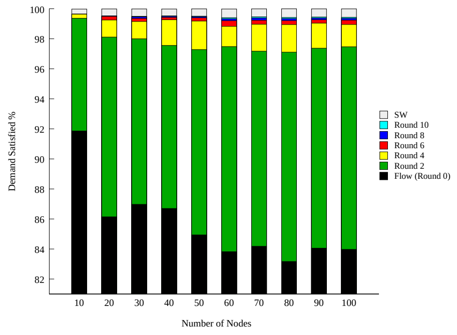


**(b)** N=60, K=30, L=35

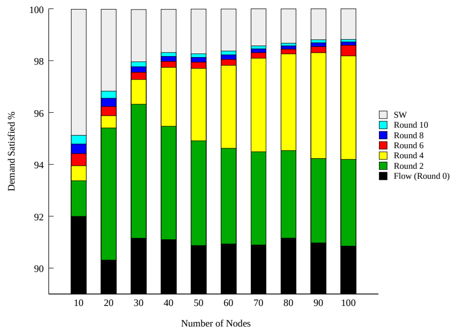


**(c)** N=60, K=60, L=150

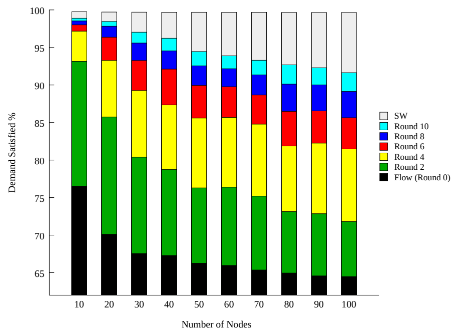
**Figure 3.9:** Plot shows improvement obtained by our scheme when presented with instances generated using the different shuffle methods. Every data point was obtained by averaging over 10 runs. SW is the solution value achieved by the old method.



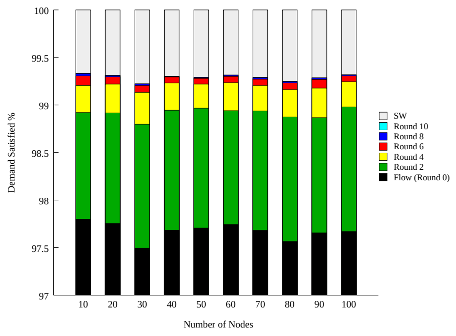
**(a)** Shuffle Method 1, K=60, L=150



**(b)** Shuffle Method 2, K=60, L=150



**(c)** Shuffle Method 3, K=60, L=150



**(d)** Shuffle Method 4, K=60, L=150

**Figure 3.10:** Performance of our scheme with varying number of nodes for different shuffle methods. The number of nodes N varied from 10 to 100. Every data point was obtained by averaging over 10 runs. SW is the solution value achieved by the old method.

| N | Our Scheme | | Old Scheme | |
|---|---|---|---|---|
| | Rounds | Demand % | Rounds (avg) | Demand % |
| 10 | **10** | 99.64 | **104.8** | 99.98 |
| 20 | **10** | 99.52 | **111.8** | 99.99 |
| 30 | **10** | 99.50 | **121** | 99.99 |
| 40 | **10** | 99.53 | **121.9** | 99.98 |
| 50 | **10** | 99.51 | **125.9** | 99.99 |
| 60 | **10** | 99.41 | **128.2** | 99.98 |
| 70 | **10** | 99.46 | **128.5** | 99.99 |
| 80 | **10** | 99.42 | **135.4** | 100 |
| 90 | **10** | 99.45 | **138.6** | 99.99 |
| 100 | **10** | 99.43 | **136.2** | 99.99 |

**(a)** Shuffle method 1

| N | Our Scheme | | Old Scheme | |
|---|---|---|---|---|
| | Rounds | Demand % | Rounds (avg) | Demand % |
| 10 | **10** | 95.12 | **103.3** | 99.98 |
| 20 | **10** | 96.82 | **109.4** | 99.98 |
| 30 | **10** | 97.96 | **119.4** | 99.97 |
| 40 | **10** | 98.31 | **119.7** | 99.98 |
| 50 | **10** | 98.26 | **124.7** | 99.99 |
| 60 | **10** | 98.37 | **127.4** | 100 |
| 70 | **10** | 98.57 | **129.3** | 100 |
| 80 | **10** | 98.67 | **135.3** | 100 |
| 90 | **10** | 98.81 | **134.4** | 100 |
| 100 | **10** | 98.82 | **137.5** | 100 |

**(b)** Shuffle method 2

| N | Our Scheme | | Old Scheme | |
|---|---|---|---|---|
| | Rounds | Demand % | Rounds (avg) | Demand % |
| 10 | **10** | 98.89 | **126.2** | 99.75 |
| 20 | **10** | 98.46 | **133** | 99.71 |
| 30 | **10** | 97.03 | **138.8** | 99.7 |
| 40 | **10** | 96.22 | **144.1** | 99.68 |
| 50 | **10** | 94.45 | **146.5** | 99.69 |
| 60 | **10** | 93.89 | **149.4** | 99.67 |
| 70 | **10** | 93.29 | **149.8** | 99.68 |
| 80 | **10** | 92.69 | **154.6** | 99.66 |
| 90 | **10** | 92.29 | **152.8** | 99.65 |
| 100 | **10** | 91.63 | **155.9** | 99.66 |

**(c)** Shuffle method 3

| N | Our Scheme | | Old Scheme | |
|---|---|---|---|---|
| | Rounds | Demand % | Rounds (avg) | Demand % |
| 10 | **10** | 99.33 | **128.3** | 100 |
| 20 | **10** | 99.31 | **139.6** | 100 |
| 30 | **10** | 99.22 | **148.9** | 100 |
| 40 | **10** | 99.30 | **159.2** | 100 |
| 50 | **10** | 99.29 | **166.6** | 100 |
| 60 | **10** | 99.32 | **170.8** | 100 |
| 70 | **10** | 99.29 | **178.7** | 100 |
| 80 | **10** | 99.25 | **183.6** | 100 |
| 90 | **10** | 99.29 | **190.3** | 100 |
| 100 | **10** | 99.32 | **196.1** | 100 |

**(d)** Shuffle method 4

**Table 3.2:** Comparison of old scheme with our scheme for various number of nodes N=10 to 100, K=60, L=150 and various shuffle methods.

# Chapter 4

# Decentralized Data Placement and Reconfiguration

This is joint work with Indrajit Bhattacharya and Srinivasan Parthasarathy. These results also appeared in [22].

## 4.1 Introduction

Distributed Hash Table (DHT) based peer-to-peer systems such as CAN, CHORD, Pastry, and Tapestry [144, 164, 149, 179] support a basic abstraction: the lookup. Given a query for a specific *key*, the lookup efficiently locates the network node which owns the key. Although all DHTs implement the basic lookup functionality efficiently, most real-life applications demand more. For instance, consider an Information Retrieval (IR) application where nodes export a collection of text documents. Each document is characterized by a $d$-dimensional vector. The field of IR is replete with vector-space methods for such document representations (for e.g, see [19, 49]). A user query consists of a vector and the user needs all documents in the database which match this vector or which are *semantically related* to it.

DHTs do not support information retrieval applications like the one above. The fundamental reason which renders DHTs ineffective in these situations is that data objects in a DHT are distributed uniformly at random across the network nodes. While this ensures that no node stores too many objects, it also scatters semantically related objects across the network. Thus, when a query is issued, the only way the DHT can return all objects relevant to it would be to flood the entire network, leading to unacceptable network loads.

Our focus in this work is to efficiently support *similarity queries for text information retrieval in DHT based overlay networks.* We introduce a new query model where users issue queries of the form $(x, \delta)$. Here $x$ is a data object and $\delta$ is a distance measure. The search algorithm needs to return all data objects $y$ in the network such that $f(x, y) \leq \delta$, where $f$ is an application specific *distance function.* The schemes presented in this chapter are geared toward the Cosine distance metric which is defined as follows: $f(x, y) = \cos^{-1} \frac{x \cdot y}{|x||y|}$, where $x \cdot y$ is the dot product between the vectors and $|\cdot|$ is the Euclidean ($l_2$) norm. The Cosine distance is a widely used distance function in text retrieval applications.

The key technical challenges which we attempt to address here are as follows:

1. Developing an efficient object placement and search mechanism such that given a query for an object, the search returns all objects in the DHT which are similar to the query object.

2. Developing efficient mechanisms for adaptive replication of popular objects so that query loads are uniformly distributed across network nodes. This is particularly relevant for systems which support similarity searching: similar objects tend to be co-located with each other and if an object is popular, then other objects similar to it can also be expected to be popular.

3. Developing mechanisms which are oblivious to the underlying DHT technology so that the resultant system can be implemented over a variety of DHT topologies, making it possible to leverage other advantages specific to each DHT.

The techniques developed in this work address the challenges identified above. In particular, we view the following as the main contributions of our work.

- We develop an indexing scheme which clusters data such that a group of closely related objects belong to a small set of clusters. This in turn paves the way for an efficient search mechanism for answering similarity queries.

- Our indexing scheme decouples object and node location in the DHT, allowing popular objects to be adaptively replicated in the DHT. We propose simple adaptive replication and randomized lookup algorithms to exploit this feature of our indexing algorithm. Our adaptive replication scheme ensures that the number of copies of a key in the DHT is proportional to its popularity and the randomized lookup scheme guarantees that a query is equally likely to be served by any of the replicas of that key in the DHT. Thus, the replication and randomized lookup algorithms together guarantee perfect load-balancing.

- We present precise analytical guarantees for the performance of our algorithms in terms of search accuracy, cost, and load-balancing. All the algorithmic and analytical results presented here are oblivious to the underlying DHT topology, thus making it possible for implementation over any DHT.

The key driver behind our techniques is the notion of *similarity preserving hash functions* (SPHs) [28]. SPHs provide a powerful and interesting property in the context of our work. Given a set of points which are at a small distance from each other, with high probability an SPH maps these points into a "small" set of related indices. Such a mapping of data objects onto indices leads to a simple search strategy as follows: a node $u$ which has a query $(x, \delta)$, computes the set of indices which are relevant to object $x$; $u$ then queries all the nodes which own these indices. The queried nodes return the set of relevant objects back to $u$. The use of SPHs for developing provably good similarity search algorithms is one of the key innovations of this work.

The rest of the chapter is organized as follows. We survey related work in Section 4.2. Section 4.3 formally describes our data and query model and Section 4.4 presents a detailed description of the major techniques developed in this work. We discuss potential extensions and optimizations of our techniques in Section 4.5. Section 4.6 presents analytical performance evaluation of our schemes and Section 4.7 presents the results of our experimental studies. We close with concluding remarks and directions for future work in Section 4.8.

## 4.2 Related Work

Several researchers have proposed mechanisms to extend the scope of DHTs beyond the traditional lookup. Vahdat *et al.* [147], Liu *et al.* [114], and Shi *et al.* [161] address efficient keyword searching in DHTs. The work of Gupta *et al.* [77] and Schmidt *et al.* [153] use SPHs to distribute high dimensional data vectors on top of a CHORD overlay. The former supports approximate range queries while the latter supports exact range queries. The work of Gopalakrishnan *et al.* [21] supports efficient set intersection operations using view trees. The `pSearch` system [167] comes closest to our work since this is the only system, prior to our work, which supports similarity searching in any DHT.

### 4.2.1 The `pSearch` System

The `pSearch` system [167] supports similarity queries for real-valued data vectors for the Cosine distance metric (see Section 4.3). It is built on top of the CAN DHT [144] and uses Lexical Semantic Indexing (LSI) [49] for indexing text documents. Object coordinates derived from these indices are used for routing and object location. Although the basic goals of `pSearch` is the same as ours, the techniques presented here differ significantly from those developed in `pSearch`. We outline some of the key differences and the resulting trade-offs below.

1. `pSearch` uses projections of object coordinates derived from the LSI algorithm as object indices. This restricts object indices to only real vectors which makes it implementable only on the CAN DHT and precludes its implementation over other popular DHTs such as CHORD, Pastry and Tapestry. This is due to the fact that all DHTs invoke customized object-to-node mapping functions for mapping a given object onto a node in the DHT. While the range of this mapping function is a real vector for CAN, this is not the case in other DHTs (for CHORD it is a real number in the range $[0, 360)$; for Pastry or Tapestry it is a bit string). In contrast, our indexing scheme simply partitions the data into clusters and assigns the same index to each object in the cluster; we allow the underlying DHT mapping functions to assign indices to nodes in the DHT. Thus, our schemes can be implemented over any underlying DHT topology.

2. The object placement algorithm in `pSearch` converts the CAN physical overlay into a semantic overlay such that nodes within a small physical neighborhood store data objects which are similar to each other. This has an implicit advantage since similar objects can be retrieved by flooding a small neighborhood of nodes within a region. This physical locality is not always possible with our object location technique, since we allow the underlying DHT mapping functions to assign object clusters to nodes. Hence, a single-hop neighborhood query in the *pSearch* system may correspond to a DHT lookup in our scheme resulting in

increased network traffic. However, the tight coupling of object and node location in *pSearch* virtually makes it impossible for adaptive replication of highly popular objects in the system for effective query load balancing. Our indexing scheme provides for adaptive replication of popular clusters and avoids hotspots; this is one of most significant flexible feature offered by our techniques vis-a-vis the `pSearch` system.

3. Our indexing scheme relies on the notion of Similarity Preserving Hash (SPH) functions which hash a group of related objects onto a small set of indices. While the primary motivation for our work is supporting cosine similarity queries for use in text retrieval applications, the basic techniques developed here can be generalized to a large class of data and query models and similarity metrics which support SPHs. One such important category is image and multimedia retrieval which can be supported by the SPHs developed by Indyk *et al.* [89]. The use of SPHs also allows us to model the behavior of our system in terms of the search accuracy vs. search cost using precise analytical models which are independent of the underlying DHT. Thus, unlike the `pSearch` system, the use of SPHs make our techniques applicable to a variety of data models and similarity metrics, allows for precise analytical modeling, and is oblivious to the underlying DHT topology, thus staking a strong claim for widespread acceptance in peer-to-peer database applications.

### 4.2.2 Adaptive Replication

Object placement algorithm within DHTs typically place objects uniformly at random on one of the DHT nodes in an attempt to balance query load. While this is reasonable under assumptions of uniform query-rate for all objects, in practice, query behavior tends to follow very skewed zipf-like distributions [20]. This behavior is even more acute in systems which co-locate related objects to support similarity searching, since objects close to popular objects also tend to be popular. Most DHTs provide only for static replication where each object in the DHT is replicated a *fixed* number of times and hence do not deal with non-uniform query distributions.

The Lightweight Adaptive Replication (LAR) protocol of Gopalakrishnan et al. [72] addresses this problem by measuring the load on individual servers and using the load measurements to create appropriate number of copies of a key. They also modify the DHT lookup primitive by augmenting nodes in the DHT with information about the newly created copies. We note that the adaptive replication technique presented in this work is similar in spirit to this scheme, since our technique also relies on server load information for spawning and retracting copies of a key. However, our scheme differs from that of LAR in significant ways: a query node in our scheme is required to have a good estimate of the current number of copies of a key in the system (failing which the query may incur more than a single DHT lookup; however the query is still guaranteed to be successful even without a good estimate). However, unlike LAR, our scheme does not require nodes in the DHT to be augmented with “routing hints to direct the lookups to the appropriate replicas. We also note that, like LAR, our scheme is also oblivious to the underlying DHT topology and can be implemented on top of any DHT. Finally, our techniques are also interoperable with LAR or any other adaptive replication protocol specific to any DHT.

## 4.3 Background: Data and Query Model

Information Retrieval (IR) applications frequently model text documents as *term vectors*. A term vector is a vector of real numbers; coordinates in the vector correspond to *terms* and the value of each coordinate represents the relative frequency of the corresponding term within the document. In general, terms may correspond to keywords or a combination of keywords found within the documents. It is also usual to normalize the term vectors so that vectors are of unit length, in order to account for the variable sizes of the documents. Several techniques exist in the IR literature for representing documents as term vectors, most of which are variants the Vector Space Model (VSM) [19] and the Latent Semantic Indexing (LSI) schemes [49].

For the rest of this chapter, we assume that all data objects are unit vectors in a $d$-dimensional Euclidean space. Equivalently, the data objects may also be viewed as points on the surface of the $d$-dimensional unit hyper-sphere. Two objects are considered similar, if they lie close to each other on the surface of the unit sphere. Formally, let $x$ and $y$ be two objects and let

$\theta$ be the angle between them. The similarity between $x$ and $y$ is defined by the function $f$, where $f(x, y) = \cos(\theta) = x \cdot y$, the so called dot-product or the inner-product of $x$ and $y$. The distance between $x$ and $y$ is defined as the angle $\theta$. The greater the value of $f$, lesser the value of the angle $\theta$, and more similar are the objects $x$ and $y$. We note that the `pSearch` system also works with this data and similarity model. Later, in Section 4.5.1, we discuss ways to generalize many of the techniques developed in this chapter to other interesting data models and similarity metrics.

We assume that user queries of the form $q = (x, \delta)$, where $x$ is a $d$-dimensional unit vector and $0 \leq \delta \leq \frac{\pi}{2}$ is the distance measure. An object $y$ *matches* query $q$ if $y$ is sufficiently close to $x$: i.e., $\cos^{-1} x \cdot y \leq \delta$. The search *accuracy* is defined as the fraction of matching objects in the system that are returned by the search. The search *cost* is defined as the number of lookups performed by the system during the search. The algorithms presented in this chapter trade-off search accuracy with respect to the search cost.

## 4.4 Design Details

Four basic techniques underlie the mechanisms developed in this chapter. Following is a brief description of these techniques.

- The**Indexing Scheme** partitions the data-space into several clusters. Each data object is assigned an index and clustering is achieved implicitly by assigning all objects which have the same index to the same cluster. The indexing scheme guarantees that any set of objects which are sufficiently similar to each other are assigned either to the same cluster or to a small group of clusters. The indices are treated as keys by the DHT; each index is owned by some node and all objects with this index are stored by the node which owns the index.

- The **Search Algorithm** computes a set of indices S which are relevant to the given query $q = (x, \delta)$; it then performs a lookup for each index in $S$. These lookups terminate at a set of nodes, which return all objects owned by them that match the query $q$. In general, a higher search accuracy would require the algorithm to compute a larger set of indices $S$ resulting in higher search costs.

- The **Adaptive Replication** algorithm ensures that the number of copies of each key in the network is proportional to its popularity. Specifically, the number of copies of each key in the DHT is proportional to the rate at which queries arrive for this key. The creation and retraction thresholds, which are global system-wide parameters determine how aggressively copies are created or retracted in the system.

- The **Randomized Lookup** algorithm guarantees that the lookup for a specific key terminates uniformly at random at one of the copies of this key. Thus, the lookup and the replication algorithms, in tandem, guarantee that the load is balanced uniformly across all copies of all keys in the system.

In the following sections, we present the details involved in each of these techniques.

### 4.4.1 Indexing

Each data object in the peer-to-peer database is assigned an index. We now propose a hash function $h$ which takes a $d$-dimensional data object $x$ as input and computes a $k$-bit string $h(x)$ as output. The string $h(x)$ is the index of object $x$. Let $r$ be a $d$-dimensional unit vector. Corresponding to this vector, we define the binary function $b_r$ as follows:

$$b_r(x) = \begin{cases} 1 & \text{if } r \cdot x \geq 0 \\ 0 & \text{if } r \cdot x < 0 \end{cases} \tag{4.1}$$

$b_r(x)$ defines the orientation of $x$ w.r.t. $r$. This function was proposed by Charikar [28] for estimating cosine distances between points in high dimensional space. He also observed that if $r$ is chosen uniformly at random from all $d$-dimensional unit vectors, then for any two vectors $x$ and $y$, $\Pr[b_r(x) \neq b_r(y)] = \frac{\delta}{\pi}$, where $\delta = cos^{-1} \frac{x \cdot y}{|x||y|}$ is the angle between the two vectors in radians. Our hash function $h$ is parametrized by a set of unit vectors $r_1, \ldots, r_k$, each of which is chosen uniformly and independently at random from the set of all $d$-dimensional unit vectors. The hash value $h(x)$ is simply the concatenation of the bits $b_{r_1}(x), \ldots, b_{r_k}(x)$. Objects with the same index belong to the same cluster. Object $x$ is stored at the node which owns the DHT key $h(x)$.

The above hashing scheme essentially attempts to group nearby objects to indices with low hamming distance. However, there is still a reasonable chance that nearby objects can differ in some bit positions in their indices. In order to reduce the probability of this bad event from occurring, we construct $t$ hash functions $h_1, \ldots, h_t$ as described above, which yields $t$ sets of object indices. This ensures that there is a high probability of two related objects hashing onto indices with low hamming distance in at least one of these sets. We note that we can treat these sets of indices as a static replication of objects; the static replicas are also analogous to the "rolling indices in the pSearch system. Further, we show both using theoretical analysis (see Section 4.6) and using simulations (see Section 4.7) that static replication boosts the search accuracy. However, it does not address the problem of load balancing, which we deal with in Sections 4.4.3 and 4.4.4. We emphasize again that these static replicas of objects are not the same as multiple copies of the DHT keys. For the remainder of the chapter, we use the term replicas to denote the static replicas and the term copies to denote the multiple copies of a DHT key created by the adaptive replication algorithm.

### 4.4.2 The Search Algorithm

The search algorithm is parameterized by a radius $r$, which is a non-negative integer. A node $u$ which generates a query $(x, \delta)$ first computes the index $h(x)$. It then computes the set $S$ of all indices whose hamming distance from $h(x)$ is at most $r$ (i.e., the set of indices which differ from $h(x)$ in at most $r$ bit positions; note that $S$ always includes $h(x)$). Let $V$ be the set of nodes in the network which own the keys in $S$. Node $u$ queries each of the nodes in $V$ . Nodes in $V$ return all data objects which match $u$s query. How is the search radius $r$ determined? The search radius $r$ is affected by various parameters such as $k$, $t$, the query parameter $\delta$, and the desired search accuracy. Fixing all other variables, an increase in the value of $r$ would result in more objects which match the query being returned. Of course, the increased accuracy is also achieved at an increased search cost. We examine the effect of $r$ on the search accuracy and cost in Section 4.6. We note that the search algorithm may be easily extended to the case where we have $t$ static replicas of the objects in the system.

### 4.4.3 Adaptive Replication

We now present the details of our adaptive replication scheme which creates and retracts keys adaptively depending on load conditions. Keys in our system refer to indices within a specific static replication, although, in general, the replication scheme is oblivious to what the keys may refer to. Recall that in a DHT, each key $y$ is mapped onto a random value $m(y)$ using a mapping function $m$; a lookup for $m(y)$ terminates at a specific node $u$ which is said to own $m(y)$; this node stores a copy of the key $y$. Our replication scheme parameterizes the mapping function $m$ with a positive integer $i$. Specifically, let $s = m(i, y)$. Then, the node which owns $s$ is responsible for storing the $i^{th}$ copy of the key $y$. We now describe how to a create a new copy and retract an existing copy of a key in the DHT. Consider a key $y$ which currently has $l$ copies. The main invariant maintained by the replication algorithm is that the copies are contiguous, ranging from 1 to $l$: i.e., the $l$ copies are placed at nodes which own values $m(1, y), m(2, y), \ldots, m(l, y)$ respectively. The copies can be visualized as nodes of a complete binary tree, with copy $i$ being the parent of copies $2i$ and $2i + 1$. We note that the binary tree abstraction is completely implicit and there are no pointers associated with children or parents of copies in reality. All nodes in the system maintain two thresholds $r_{high}$ and $r_{low}$ and a periodic local timer. A node which owns a copy of $y$ performs the following check at the end of each period: let the number of queries it received for key $y$ in the previous time period be $q$; if $q \geq r_{high}$, it creates copies $l+1$ and $l+2$ of $y$. If $q < r_{low}$, it retracts copies $l$ and $l-1$ of $y$. Creation and retraction are both achieved by sending a message to the parent of the nodes which own the corresponding copies; if the parent has not already performed the creation (retraction) it performs this action after receiving a creation (retraction) message. These messages are routed using the standard lookup primitives of the DHT. We observe that the creator or retractor of a copy need not know the node which owns the last copy l of $y$, or its parent, but just the value of $l$.

How does a creator (or a retractor) of a copy know the value of $l$? We note that a simple solution is to notify all nodes which own a copy of $y$, whenever a copy is created or retracted in the system. Yet another solution is to perform a simple binary search in the range $1, \ldots, l_{max}$, where

$l_{max}$ is the maximum number of replicas allowed for any key within the DHT. We note that the latter discovers the value of $l$ after $O(\log(l_{max}))$ lookups with high probability.

### 4.4.4 Randomized Lookup

Let node $u$ generate a query for key $y$ and let there be $l$ copies of $y$ currently in the system. If node $u$ knows the exact value of $l$, it chooses a random number $i$ in the range $1, \ldots, l$ and performs a lookup for $m(i, y)$. This ensures that all copies of the key are equally likely to serve this query. However, in general, nodes can not be expected to have exact information about the number of replicas for a specific key in the DHT. We now show how our randomized lookup solves this problem in two scenarios. In the first scenario, node $u$ does not have any information about $l$. In this case, it performs a randomized binary search in the range $1, \ldots, l_{max}$ to obtain a copy of $y$. Specifically, $u$ selects a random number $l_1$ uniformly in the range $1, \ldots, l_{max}$ and performs a lookup for $m(l_1, y)$. If this lookup returns a copy of $y$, the lookup terminates. Else, the node repeats the randomized binary search in the range $1, \ldots, l_1 - 1$. The randomized lookup terminates whenever any of these DHT lookups returns a copy of $y$. Observe that the randomized lookup is guaranteed to terminate for any initial estimate of $l$ since the successive DHT lookups are in strictly decreasing ranges and a lookup for $m(1, y)$ is guaranteed to terminate successfully.

Fixing the initial estimate of $l$ at $l_{max}$ results in at most $O(\log(l_{max}))$ DHT lookups with high probability. However, this increased lookup latency may be unacceptable for many applications. One way to avoid this problem is for each node to estimate the value of $l$ using counting bloom filters [56]. Counting bloom filters are compact data structures for checking set membership in distributed environments. In our setting, the entries in the counting bloom filter are of the form $(i, y)$. If such an entry exist, it indicates that the $i^{th}$ copy of key $y$ exists in the system. These bloom filters are updated periodically to reflect any changes in the number of copies of any key. We note that an exact estimate for the number of copies is not required for the correctness of our lookup algorithm. Hence, one possible optimization in the bloom filter design is to just store entries of the form $(i, y)$ where $i$ is a power of two, instead of all values of $i$ in the range $1, \ldots, l_{max}$.

This results in an estimate of $l$ which is at most within a factor of two from the correct value, thus only slightly increasing the randomized lookup latency while decreasing the size of the bloom filter substantially (from $O(l_{max})$ to $O(\log(l_{max}))$). The work of Mitzenmacher [123] discusses several techniques for updating counting bloom filters in distributed settings with low message overhead.

## 4.5 Discussion

We now discuss extensions to our indexing scheme for other models and optimizations for our search algorithm in idealized hypercube-like networks such as Pastry and Tapestry.

### 4.5.1 Extensions to other data models and distance metrics

The indexing scheme exploits the central property of SPHs that objects at a small distance from each other hash onto nearby indices with high probability. While the hash functions presented here are motivated by text retrieval applications and geared toward cosine similarity, the basic indexing and search mechanisms can be extended to several other classes of data models and similarity metrics. One such important class of distance metrics is the usual Euclidean norm defined as follows: $f(x,y) = (\sum_{i=1}^{d} |x_i - y_i|^r)^{\frac{1}{r}}$ $r \geq 1$,$x, y \in \mathcal{R}^d$ In this case, $f$ is the the $l_r$-norm of the vector $(x-y)$. In practice, the most widely used norms are $l_1$, $l_2$, and the $l_{\inf}$ norms. These metrics are of significance in image and multimedia retrieval, where a common first step is to extract a set of numerically-valued features or parameters from the document [138, 55]. After these features have been extracted, an image in the database may be thought of as a point in a high-dimensional space. User queries are again transformed to the same vector space where the documents are represented and may be thought of as nearest-neighbor searches in the high-dimensional space. We note that the similarity-preserving hashing scheme of Indyk *et al.*[89] perform a clustering of the data space are geared toward the normed-distance. These hashing-schemes can be readily plugged into the our indexing mechanism thus extending our approach to image and multimedia databases as well. Yet another notable case is that of set similarity measure which is defined as follows: $sim(A,B) = \frac{|A \bigcap B|}{|A \bigcup B|}$; here $A$ and $B$ are sets and $sim(A,B)$ also known as the Jaccard coefficient in Information Retrieval literature, is a measure of the similarity between the sets $A$ and $B$. We note that one may plug in the *min-wise independent permutations* introduced by Broder *et al.*[26] to handle this similarity measure.

### 4.5.2 Routing Optimizations for Hypercube-like Networks

In an idealized scenario, Pastry and Tapestry can be viewed as implementations of a hypercube network. Each node in the overlay has a unique ID which is a $k$-bit binary string. Two nodes are overlay neighbors of each other if and only if the hamming distance of their IDs is one (i.e., they differ in exactly one bit). An object is stored by a node if the object index matches with the node ID.

In this scenario, a node $u$ with a query $(x, \delta)$ performs a single lookup for $x$ which terminates in a node $v$ whose ID is $h(x)$. Node $v$ performs a local search by flooding the query to all its $r$-hop overlay neighbors where $r$ is the search radius. These nodes return their local search results to $v$ which gathers and returns the union of all the results to $u$. Note that this optimization does not reduce the number of nodes being queried. However, it reduces the routing load substantially since each lookup is now replaced by a single hop message.

We note that the setting described here may not be realizable in practice due to two main reasons. The above optimization assumes that each key is owned by exactly one node which precludes adaptive replication. Secondly, the index size $k$ is determined during network creation when the total number of nodes is unknown. Further, the network dynamics could lead to nodes joining and leaving the system resulting in variable number of network nodes. We leave the problem of achieving the routing optimizations in the hypercube networks under realistic network scenarios as an interesting topic of future research.

## 4.6 Analysis

Consider a query $q = (x, \delta)$. Let $S$ be the set of all objects in the database which matches this query. Let $S'$ be the set of objects returned by the search algorithm. Recall that $t$ is the number of static replications, $k$ is the number of bits in the index and $r$ is the search radius. We define the accuracy of the search to be $|S'|/|S|$, i.e., the fraction of objects in the database which match the query and which are returned by the search. $E[|S'|/|S|]$ denotes the expected accuracy.

### 4.6.1 Accuracy

**Theorem 4.**

$$\mathbf{E}[|S'|/|S|] \geq 1 - \left(1 - \sum_{i=0}^{r} \binom{k}{i} \left(\frac{\delta}{\pi}\right)^{i} \left(1 - \frac{\delta}{\pi}\right)^{k-i}\right)^{t} \tag{4.2}$$

*Proof.* Consider a specific object $z$ in the DHT which matches the query $q$, i.e., $\cos^{-1}(xz) \leq \delta$. Consider a specific replication $p$ among the $t$ replications. Let $\phi(s_1, s_2)$ denote the hamming distance between two equal sized bit strings $s_1$ and $s_2$. We now compute the probability of the hamming distance between $h_p(z)$ and $h_p(x)$ being equal to a specific value $i$. Recall from Equation (4.1) that the probability of a specific bit in $h_p(z)$ differing from its corresponding bit in $h_p(x)$ is equal $\frac{\theta}{\pi}$, where $\theta$ is the angle between the two objects in radians. Hence we have,

$$\begin{aligned}
\Pr[\phi(h_p(x), h_p(z)) = i] &= \binom{k}{i} \left(\frac{\theta}{\pi}\right)^{i} \left(1 - \frac{\theta}{\pi}\right)^{k-i} \\
\Pr[\phi(h_p(x), h_p(z)) \leq r] &= \sum_{i=0}^{r} \binom{k}{i} \left(\frac{\theta}{\pi}\right)^{i} \left(1 - \frac{\theta}{\pi}\right)^{k-i} \\
\Pr[\phi(h_p(x), h_p(z)) > r] &= 1 - \left(\sum_{i=0}^{r} \binom{k}{i} \left(\frac{\theta}{\pi}\right)^{i} \left(1 - \frac{\theta}{\pi}\right)^{k-i}\right) \\
\Pr[\forall p,\ \phi(h_p(x), h_p(z)) > r] &= \left(1 - \sum_{i=0}^{r} \binom{k}{i} \left(\frac{\theta}{\pi}\right)^{i} \left(1 - \frac{\theta}{\pi}\right)^{k-i}\right)^{t}
\end{aligned}$$

Hence,

$$\begin{aligned}
\Pr[z \text{ is returned by the search }] &= 1 - \Pr[\forall p,\ \phi(h_p(x), h_p(z)) > r] \\
&= 1 - \left(1 - \sum_{i=1}^{r} \binom{k}{i} \left(\frac{\theta}{\pi}\right)^{i} \left(1 - \frac{\theta}{\pi}\right)^{k-i}\right)^{t}
\end{aligned}$$

Since the expected accuracy is the same as the above probability, and since $\theta \leq \delta$ for all objects $z$, the theorem holds. □

### 4.6.2 Search Cost

**Theorem 5.** *Let the number of keys being looked up be $C$. Then,*

$$C = t \sum_{i=0}^{r} \left( \begin{array}{c} k \\ i \end{array} \right) \tag{4.3}$$

*Proof.* For a specific replication $p$, the search algorithm lookups all indices which are within a hamming distance $r$ from the index $h_p(x)$. Since, there are exactly $\sum_{i=0}^{r} \left( \begin{array}{c} k \\ i \end{array} \right)$ such indices and since there are $t$ replications, the theorem follows. $\square$

We note that in general, the search cost could potentially be greater than $C$. This is because, each key lookup could result in more than a single DHT node lookup in the randomized lookup algorithm. However, we show in Section 4.7 that this is not the case: each key lookup on an average incurs only slightly more than one node lookup even with a bloom filter of small size.

### 4.6.3 Load Balance

**Theorem 6.** *Let the number of copies of a key $y$ at time $t$ be $l$. Then a lookup for key $l$ at time $t$ will terminate at one of these $l$ copies uniformly at random.*

*Proof.* Consider the last step of the randomized lookup algorithm when the DHT query for a copy of $y$ is successful. Let the estimate for the value of $l$ before this step be some value $j \geq l$. Since, the DHT lookup is for a random copy of $y$ in the range $1, \ldots, j$, all the copies in this range have equal probability of being looked up. Hence, the theorem holds. $\square$

## 4.7 Experiments

Our experiments assume an underlying CHORD network that provides lookup, insert and delete primitives. The number of nodes in the network is fixed throughout all our experiments.

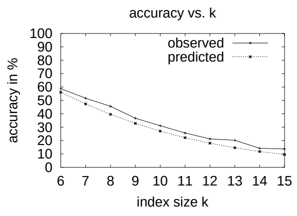
### 4.7.1 Similarity Search

Data objects in our simulations are sampled uniformly at random from the surface of the $d$-dimensional unit hypersphere. This is achieved by sampling each coordinate of the vector independently from a standard normal distribution. A $k$-bit index of a data object is created using $k$ random unit vectors. We use static replication with $t$ hash functions. We use the publicly available CHORD simulator [1] for evaluating the accuracy results. Initially all nodes and keys are inserted into the CHORD simulator. Each query object is a randomly sampled $d$-dimensional unit vector. We observe the effect of the number of replicas $t$, the search radius $r$, the size of the index $k$, the dimensionality of the data $d$, the number of nodes $n$, the number of data objects $N$, and the query parameter $\delta$ on the search accuracy and storage load. The default values of these parameters are in the table below. Results are averaged over 100 trials.

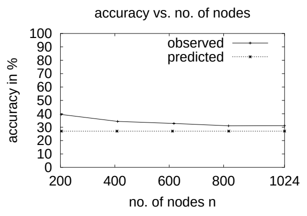
| $N$ | $d$ | $k$ | $t$ | $r$ | $\delta$ | $n$ |
|---|---|---|---|---|---|---|
| 50,000 | 15 | 10 | 1 | 1 | 0.75 | $2^k$=1024 |

**Table 4.1:** Default values for network parameters used in the similarity search experiments

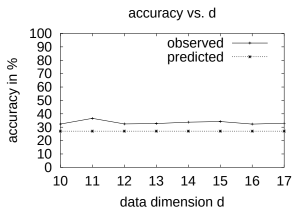
To evaluate storage load, we sort the nodes in decreasing order of number of objects they store and group them into 20 buckets. For each of the buckets, we plot the percentage of the total number of objects stored in the nodes of the bucket. The baseline for comparison is the uniform distribution where each bucket stores 5 percent of the objects. Figure 4.1 (a)-(g) plot the effect of the various system parameters on accuracy. We plot both the experimentally observed values as well as the analytically predicted ones. The accuracy increases as a function of the number of replicas $t$ and the search radius $r$. It does does not vary much as a function of the data dimension $d$ or the number of nodes $n$ or the number of data objects $N$ in the system. However, the accuracy decreases with the size of the index $k$ as well as the query parameter $\delta$. Our analysis predicts the experimental trends accurately in all the trials. This suggests that the accuracy guarantees provided by our analysis do not only hold in expectation, but also with high probability. Also note that the experimentally observed values are always higher than the analytically predicted ones. This is explained by the fact that our analysis always yields a lower bound on the expected accuracy rather than the exact value. Figure 4.1 (h)-(i) plot the effect of the number of replicas and the size of the index on the storage load across nodes. We observe that increasing the size of the index $k$ adversely affects the storage load balance while increasing the number of replicas $t$ aids load balance. Varying other parameters does not seem to change the storage distribution across the nodes.



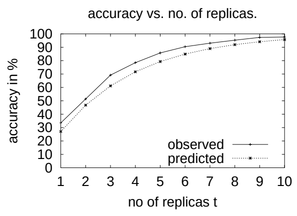
**(a)**



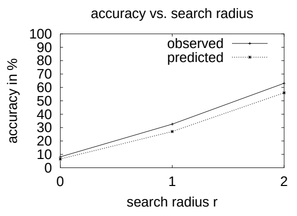
**(b)**



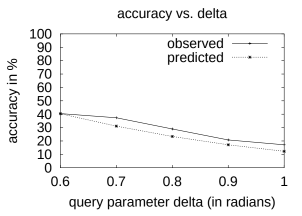
**(c)**



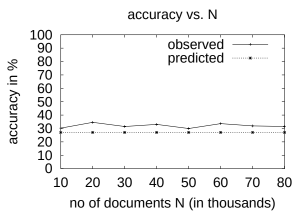
**(d)**



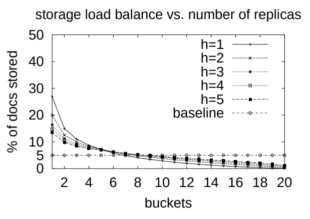
**(e)**



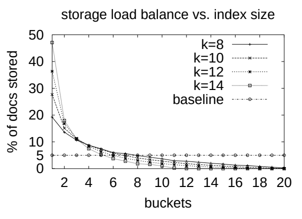
**(f)**



**(g)**



**(h)**

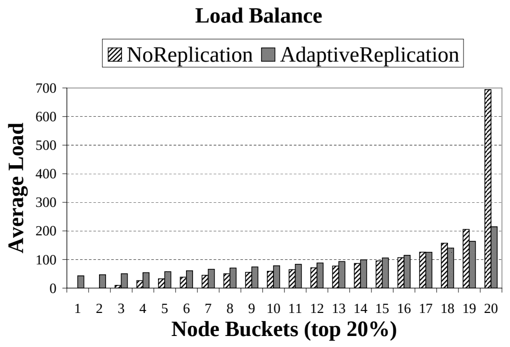


**(i)**

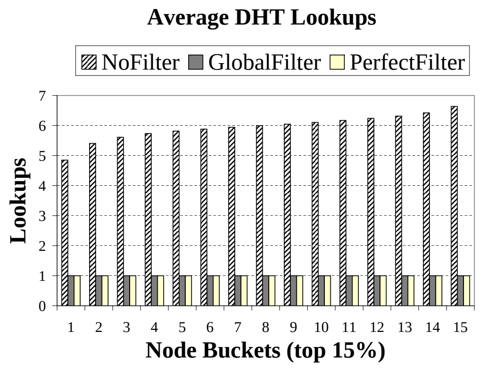
**Figure 4.1:** Experimental Results for Accuracy and Load Balance

### 4.7.2 Adaptive Replication

In our adaptive query load balancing experiments, we have 100,000 data objects distributed over a network with 5000 nodes. The data and keys are generated as mentioned in the previous section. We generate 100,000 queries for these objects according to a Zipf distribution. The skew in



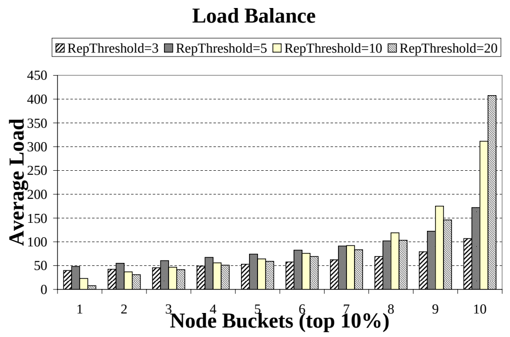
**(a)** Effect of Adaptive Replication on Load Balance.



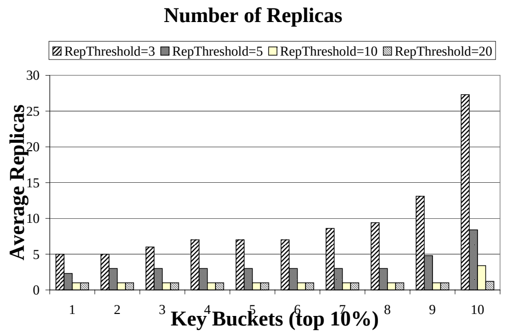
**(b)** Average Number of DHT Lookups with Global Bloom Filter, Perfect Knowledge and Randomized Binary Search.

**Figure 4.2:** Experimental Results

the distribution causes a small number of keys in the network to become load hot-spots while most other keys receive very few queries. The queries arrive according to an exponential distribution with an expected interarrival time of 1 time unit . Local timer events occur every 1000 time-units. All nodes maintain a query log (since the last local timer event) for each copy of a key assigned to it. At a local timer event, a node calculates the query rates for each of the keys assigned to it and then decides for each key whether to create a copy or retract an existing copy. This decision in determined by global creation and retraction thresholds. We study the effect of load balance with

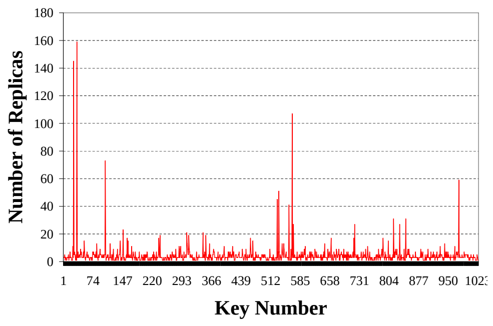


**(a)**

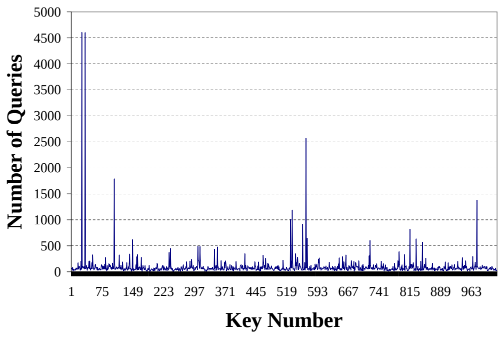


**(b)**

**Figure 4.3:** Effect of Varying Replication Threshold



**(a)** Replication for keys



**(b)** Query distribution over keys

**Figure 4.4:** Replication Response to Query Rate

respect to the thresholds. In all our experiments, retraction is turned off ($r_{low} = 0$). This also makes analyzing the results easier. We set $l_{max}$, the maximum number of copies of a key to 250.

When querying a key $y$, a node needs to estimate the number of active copies of $y$. The query converges to an active copy using randomized binary search over the range $[1, est(y)]$, where $est(y)$ is the estimated number of current copies of $y$. This range determines the number of DHT lookups before the query terminates. We compare two different schemes for estimating the number of copies. First, we use a global counting bloom filter that has 4-bit counters, 2 hash functions and whose size is $3 \times 2k \times l_{max}$, where $k$ is the size of each key (10 in our case). This bloom filter experimentally generates a false positive rate of 0.237. We compare this with the pessimistic estimate $est(y) = l_{max}$. We also compare with the ideal (hypothetical) scenario where every node has perfect knowledge about the number of copies for each key.

In Figure 4.2a and 4.3(a) respectively, we compare load balance across nodes with and without adaptive balancing and with different creation thresholds. Both figures show the top 20 percent of the nodes that have the highest loads. Clearly, the load on the hot-spot nodes is alleviated by spreading the load over other nodes. It can be observed that nodes that had low load with no adaptive replication have progressively higher loads with more aggressive copy creation. However, the load balancing does not come for free. We can see from Figure 4.3(b) that the lower the threshold, the more the number of copies created, which explains the reduced load on hot-spots.

It can also be seen from the plot that the number of copies created is not uniform over the keys. Some keys have significantly more copies than others for all thresholds. Ideally, we would like the popular keys to have more copies. This is confirmed by Figure 4.4. Figure 4.4(b) shows the query rate over the 1024 keys in the network. Note the significant spikes which denote the popular keys. Figure 4.4(a) shows the number of copies created for the same keys when using the most aggressive threshold of 3. We can see that the replica creation correlates with query rates. Specifically, the following table presents the correlation coefficient of the distribution of requests over keys and the distribution of number of copies created for the keys for different creation thresholds. We can see that in all cases the distributions are very strongly correlated, and the more aggressive the creation threshold, the stronger the correlation.

| $thresh$ | 3 | 5 | 10 | 20 | 50 |
|---|---|---|---|---|---|
| CorCoeff | 0.975 | 0.971 | 0.939 | 0.899 | 0.724 |

**Table 4.2:** Correlation Coefficients between query rate on keys and number of replicas created for key when using different creation thresholds. 1.00 indicates perfect correlation

In Figure 4.2b, we compare the average number of DHT lookups for a query on a key when estimating the current number of copies for the key using a bloom filter with the pessimistic estimate of $l_{max}$ and the hypothetical scenario when the correct number of copies in known. We observe that the use of the bloom filter with a small false positive rate reduces the number of lookups to 1 per query, which is the case for perfect knowledge. However, without the bloom filter, the average number of lookups is about 5.

## 4.8 Conclusion

We have presented a framework for indexing and searching data objects in peer-to-peer information retrieval systems. Our schemes use SPHs to map semantically related data objects to a small set of indices leading to a simple and efficient search algorithm. Our indexing algorithms decouple object characteristics and node locations, thus enabling adaptive replication of popular indices resulting in efficient load-balancing. Our framework is oblivious to the underlying DHT network topology and can be implemented on a wide variety of structured overlays such as CAN, CHORD, Pastry and Tapestry. Plans for future work include extensive experimental evaluation of the search and replication techniques in realistic distributed environments, evaluation of our schemes with data sets obtained from real applications and comparing our results with the pSearch system.

# Chapter 5

# Data Monitoring (One-shot Queries)

This is joint work with Supratim Deb, K. V. M. Naidu, Rajeev Rastogi and Anand Srinivasan. These results also appeared in [104].

## 5.1 Introduction

Many large-scale distributed applications require aggregate statistics (e.g., MIN, MAX, SUM, AVERAGE) to be computed over data stored at individual nodes. For example, in peer-to-peer systems [149, 164], the average number of files stored at each peer node or the maximum size of files exchanged between nodes is an important input to system designers for optimizing overall performance. Similarly, in sensor networks [142, 116], disseminating individual readings of temperature or humidity among all the sensor nodes, besides being too expensive, may also be unnecessary, and aggregates like MAX or AVERAGE may suffice in most cases. And finally, in a wireless network monitoring application using software probes deployed on mobile handsets to monitor performance, a service provider may be more interested in the abnormal measurements recorded by the probes like unusually low signal strength or atypically high application response times.

Depending on the application, the aggregate computation procedure must satisfy some of the following requirements.

- *Scale to a large number of nodes.* P2P systems and sensor networks can have millions of participating nodes. The procedure should be able to handle such massively distributed applications, and overall computation times should increase gradually and smoothly as new nodes join.

- *Be robust in the presence of failures.* Link and node reliability can be expected to be poor in wireless and sensor networks. Thus, the procedure must be resilient to node failures and message loss.

- *Incur low communication overhead.* Wireless links typically have low bandwidths, and in sensor networks, nodes have limited battery lives. As a result, the computation process should involve only a small number of message transmissions.

While the exact trade-off between the aforementioned requirements is not completely understood, an important question is whether it is possible to design efficient algorithms satisfying a sizable subset of the above-mentioned requirements.

For example, consider a centralized approach in which each node transmits its value to a central coordinator that then computes the aggregate. Clearly, this is extremely efficient in terms of message overhead. However, it is lacking in terms of scalability and reliability since the coordinator can quickly become a bottleneck and is a single point of failure. Similarly, though alternate approaches based on propagating aggregate computation up the nodes of a deterministic tree solve the scalability issue [80, 146], they are still susceptible to node and link failures.

To overcome the above-mentioned scalability and reliability problems, several researchers have proposed decentralized gossip-based schemes for computing various aggregates in overlay networks [107, 129, 93, 30, 23]. In gossip-based protocols, each node exchanges information with a randomly-chosen communication partner in each round. By their very nature, gossip-based schemes are robust; they are resilient to message failures as well as node failures, thus making them ideally suited for P2P, wireless and sensor networks with potentially poor link-reliability.

Much of the early gossip work focused on using randomized communication to propagate a single message throughout a network of $n$ nodes [50, 140, 101]. More recently, Kempe *et al.* [107] presented the first set of analytical results on computation of aggregate functions using randomized gossip. They analyzed a simple gossip-based protocol for computing sums, averages, quantiles and other aggregate functions. In their scheme for estimating averages, each node selects another random node to which it sends half of its value; a node on receiving a set of values just adds them to its own halved value. Kempe *et al.* showed that these values converge to the true average in

$O(\log n)$ rounds resulting in $O(n \log n)$ messages.

In this chapter, we address the following question: *is it possible to reduce the message complexity of aggregate (max, sum, average, rank of an element) computation schemes from* $O(n \log n)$ *while relaxing the number of rounds to slightly exceed* $\log n$? We present a novel scheme to compute MIN, MAX, SUM, AVERAGE and RANK using $O(\log n \log \log n)$ rounds of communication and $O(n \log \log n)$ messages. To the best of our knowledge, ours is the first result that computes these various aggregates in a network with probabilistic link and node failures using only $O(n \log \log n)$ messages. Thus, compared to previous work [107], our scheme achieves a significant reduction in communication overhead at the cost of only a modest increase in the number of rounds. This can yield significant benefits in terms of lowering congestion and lengthening node lifetimes in bandwidth and energy-constrained environments like wireless and sensor networks.

Our algorithms achieve this $O(\log n / \log \log n)$ factor reduction in the number of messages by randomly clustering nodes into groups of size $O(\log n)$, selecting representatives for each group, and then having the group representatives gossip among themselves. It is interesting to note that Karp *et al.* [101] proved that a single message cannot be spread in a network using less than $n \log \log n$ message exchanges for a class of algorithms referred to as *address-oblivious* algorithms. Although our algorithm is not "strictly" address-oblivious, this lower-bound result indicates that it might be hard to reduce the message complexity further without substantially increasing the number of rounds.

The rest of this chapter is organized as follows. In Section 5.2, we present the underlying assumptions of our gossip framework. Section 5.3 describes prior related work. In Section 5.4, we describe our schemes for computing the various aggregates. Finally, we conclude in Section 5.5.

## 5.2 Model

The network consists of a set $V$ of $n$ nodes; each node $u \in V$ has a value denoted by $val(u)$. We are interested in computing aggregate functions like MIN, MAX, SUM, AVERAGE, RANK etc. of the node values.

The nodes communicate in discrete time-steps referred to as *rounds*. As in prior work on this problem [107, 101, 23], we assume that communication rounds are synchronized, and that all nodes can communicate simultaneously during a given round. The communication graph can be either *push-based* or *pull-based*. In the push-based model, a node selects a communication partner at random, and transmits information. A node can transmit to only one node in a round. In the pull-based model, a node chooses a communication partner at random and requests information. Thus, in this model, a node can receive from only one node in a round.

We assume that each node can communicate with every other node; each node chooses a communication partner independently and uniformly at random. A node $u$ is said to *call* a node $v$ if $u$ chooses $v$ as a communication partner. Once a call is established, we assume that information can be exchanged in both directions along the link.

Message sizes are bounded by $O(\log n + \log q)$ bits, where $\{1 \ldots q\}$ is the range of values at the nodes. The values at the nodes do not change during the execution of a query. When multiple nodes attempt to communicate with a node, then a connection is either queued up or rejected (if the queue size is already sufficiently large).

We assume the failure model of [107]. In particular, we assume two types of failures: **(i)** some fraction of the nodes may crash initially, and **(ii)** links are lossy and messages may get lost. Thus, while nodes cannot crash during the execution of the algorithm, communication can fail (either due to lossy links or due to initial node crashes) with a certain probability $\delta$. W.l.o.g., we assume that $\delta$ is some constant such that $\frac{1}{\log n} < \delta < \frac{1}{8}$. Our results can also be proved without this assumption. In particular, for larger values of $\delta$, we can make $O(1/\log(1/\delta))$ repeated calls to bring down the call failure probability below $\frac{1}{8}$. On the other hand, call failure probabilities lower than $\frac{1}{\log n}$ only make it easier to prove our claims.

We consider two query models: one in which only a single node that initiates the query is interested in knowing the result, and another in which all nodes need to know the query answer.

## 5.3 Related Work

Randomized gossip-based schemes for spreading a single message update in a network date back to the work on epidemic algorithms by Demers *et al.* [50]. Initial work on spreading a single message using randomized gossip [50, 140] essentially show that a single message can be spread in a network of $n$ nodes in $O(\log n)$ rounds and $O(n \log n)$ message transfers. Karp *et al.* [101] presented an improved algorithm and showed that even when a $\delta$ fraction of the nodes and messages can fail adversarially, all but a $O(\delta)$ fraction of the nodes in the network will have the message within $O(\log n)$ rounds and using only $O(n \log \log n)$ messages. They also gave lower bounds for the problem of single message dissemination.

Several works have considered the problem of deterministic in-network aggregation using trees [80, 146]. As shown in [107] and [23], such approaches are not resilient to node and message failures.

Kempe *et al.* [107] used gossip to compute aggregates of a distributed collection of values. They presented schemes that compute the sum and average of a distributed collection of values in $O(\log n)$ rounds and $O(n \log n)$ messages. They also extended the scheme to compute rank, select random samples, quantiles (using $O(\log^2 n)$ rounds and $O(n \log^2 n)$ messages) and several other aggregate functions. Our work aims to save an $O(\log n/ \log \log n)$ factor in the number of messages used to compute these aggregates while giving up a $O(\log \log n)$ factor in the number of rounds.

Boyd *et al.* [23] considered non-uniform gossip where the probability of node $i$ communicating with its neighbor $j$ is $P_{ij}$. Their proposed algorithm is different from the standard uniform gossip model in that it considers an asynchronous setting and in each asynchronous clock tick, it finds a random matching between the vertices. These vertices then average and update their values. They also presented a distributed scheme to find the optimal communication probabilities for each pair of vertices to ensure that the gossip algorithm converges at the fastest rate.

Finally, in [30], the authors employ a gossip-based approach to compute aggregates in a wireless sensor network setting. They present an algorithm with better performance based on a property of wireless transmissions where all nodes within the radio range can hear a broadcast.

## 5.4 Our Scheme

In this section, we describe our various schemes; the scheme for computing MAX is detailed in Section 5.4.2, and the schemes for computation of SUM, AVERAGE and RANK are described in Section 5.4.3. However, before delving into the details of our approach, we discuss some of our initial attempts that did not work.

### 5.4.1 Simple Approaches (that do not work)

In this subsection, we discuss two approaches that are simple, but have high message complexity. These will motivate the need for a more sophisticated solution.

Repeated rumor-spreading Computation of MAX is in some sense similar to the rumor-spreading problem considered in [101]; however, we cannot simply invoke their results to spread the MAX value throughout the network because each node in the network holds a potentially important piece of information. Thus, naively running the rumor-spreading algorithm of [101] by considering each node's information as a rumor imposes a communication cost of $O(n^2 \log\log n)$.

Random query trees Another attempt to compute MAX is to gossip for $O(\log\log n)$ rounds, which will result in the MAX spreading to $O(\log^p n)$ nodes (for some constant $p$) with only $O(n \log\log n)$ work. Then, the query node selects two nodes at random from the set of all nodes, and each selected node then repeats this process. A selected node marks itself to ensure it is not picked again. This construction goes on until the tree has $O(n/\log^{p-1} n)$ nodes. Once this happens the nodes will aggregate values up the tree and the query node will have the MAX w.h.p. (with probability at least $1 - O(n^{-\alpha})$ for some constant $\alpha > 0$) after $O(\log n)$ rounds. This is because the probability that the tree does not contain the MAX is at most $(1 - \frac{\log^p n}{n})^{n/\log^{p-1} n} \leq 1/n$.

However, consider the communication complexity of this scheme. Consider the penultimate level of the tree. We have about $n/c\log^{p-1} n$ nodes at this level. Each of them needs to contact two *new* neighbors in the current round. The probability of a failed call for a node at this level is $\frac{1}{c\log^{p-1} n}$. To ensure this is $O(n^{-\alpha})$, each node has to draw $O(\log^p n/\log\log n)$ random samples.

This means that, at the last level alone, the scheme has communication complexity $O(\frac{n}{\log^{p-1} n} * \frac{\log^p n}{\log \log n}) = O(n \log n / \log \log n)$. Note that we are ignoring the fact that there might be collisions between the samples chosen at the same level. This however only makes the tree smaller and our argument stronger. Further, if we consider message failures as well, the communication overhead will be even higher.

### 5.4.2 Computation of MAX

We first describe the idealized version of our scheme to give the key intuition, and subsequently present the more practical version.

#### Intuition

Suppose that, incurring zero cost, we can divide all the nodes into $\frac{n}{\log n}$ groups, each of size $\log n$, and each group having a fixed root (let us call the roots *red* nodes and the others *blue* nodes). Now each red node can determine the MAX in its group, for example, by sequentially getting values from all nodes in the group and computing MAX. Note that this will need $O(\log n)$ rounds and $O(\log n)$ messages per group. Let us call this phase as *Grouping.*

Next, the red nodes gossip among themselves to compute MAX. For this, one can use the scheme in [107] to compute MAX for $m$ nodes in $O(\log m)$ rounds and $O(m \log m)$ messages. Since, there are $\frac{n}{\log n}$ red nodes involved in gossiping, we get a complexity of $O(\log n)$ rounds and $O(n)$ messages. We refer to this phase as *Gossip.*

Finally, the red nodes propagate the MAX in their own groups, which has complexity similar to the Grouping phase. We call this phase as *Sampling/Propagation.* Essentially, if any node wishes to know MAX, it can do sampling, but, if all the nodes wish to know MAX, then few nodes can do sampling followed by propagation to the other nodes.

Note that this gives us an *ideal* scheme with $O(\log n)$ rounds and $O(n)$ messages. However, in order to achieve it in the presence of node and message failures, the Grouping phase must be performed in a distributed manner . Deterministic grouping is not possible due to initial node failures, which can potentially result in a red node failure if chosen deterministically. Therefore, we propose a randomized strategy to form the groups. Each node decides to be a red node independently with probability $\frac{1}{\log n}$. This gives roughly $O(\frac{n}{\log n})$ red nodes to start with. Now, the red nodes start forming groups by randomly contacting/being contacted by other nodes. This group formation happens in two phases: first *Push* and then *Pull*. Essentially, the Push and Pull phases simulate the *ideal* case, but in a distributed manner. Below, we describe these further in the context of our overall algorithm, and also argue that it is necessary to do both push and pull for making every node part of some group.

## Overview of the Algorithm

With the above intuition in place, we are now ready to provide an overview of our scheme. The scheme consists of four phases: Push, Pull, Gossip and Sampling/Propagating.

1. **Push:** Initially all nodes are unmarked (no color assigned). Roughly $\frac{n}{\log n}$ decide to be red nodes. Each red node makes $O(\log n \log\log n)$ requests for other nodes to join. Each non-red node accepts one of the join requests randomly and marks itself blue. Also, each successful join updates the value at the red node to the max of the two values. At the end of this phase, at most $O(\frac{n}{\log n})$ nodes remain unmarked and each red node knows the MAX of its current group. The complexity of this phase is $O(\log n \log\log n)$ rounds and $O(n \log\log n)$ messages. Clearly, at the end of this phase, group size is at most $O(\log n \log\log n)$.

2. **Pull:** Each remaining unmarked node makes $O(\log n)$ calls and joins the first group it successfully calls (*i.e.*, the call is not rejected by the red node of the group). In each round, a red node accepts at most $O(\log\log n)$ calls and drops the remaining calls, so that its group size is at most $O(\log n \log\log n)$ (we will see in a moment why this is needed). At the end of this phase, all nodes are marked with some color w.h.p., and thus, part of some group. Also, each red node knows the MAX of its current group. The complexity of this phase is $O(\log n)$ rounds and $O(n)$ messages.

3. **Gossip:** Once grouping is done, the red nodes start the Gossip phase where they gossip among themselves. Since calls are made uniformly and randomly, a red node might end up calling a blue node. We can easily fix this by having the blue node return its parent red node to the caller so that it can be called in the subsequent round. Note that this does not make much difference to the protocol except for increasing the call failure probability. Also, since groups are not of the same size and the probability of a red node receiving a call is directly proportional to its group size, this is no longer uniform gossip, as considered in [107]. Using the fact that the group sizes are $O(\log n \log\log n)$, we show that at the end of this Gossip phase (after each red node has made $O(\log n)$ calls), the number of nodes with the MAX is $\Omega(n/(\log n \log\log n))$. The complexity of this phase is $O(\log n \log\log n)$ rounds and $O(n)$ messages.

4. **Sampling/Propagation:** Any node that wishes to compute MAX requests $\log n$ other nodes to "sample" $O(\log n \log\log n)$ nodes for the MAX. The maximum of these $O(\log^2 n \log\log n)$ samples is then reported as the MAX. We show that this is just the right number of samples for a node to get the MAX w.h.p. The message complexity of this phase is poly($\log n$).

   The preceding shows how any node interested in MAX can do sampling to obtain the MAX. If the MAX needs to be propagated to all the nodes, roughly $\Theta(\log n)$ nodes (this can be achieved by each node tossing a coin with probability $c \log n/n$) decide to be propagators and sample the max values. The result is then propagated to all the nodes using a modified version of the rumor spreading algorithm of [101]. This requires $O(\log n)$ rounds and $O(n \log\log n)$ messages.

Note that we have two phases (a Push phase followed by a Pull phase) during the group construction. This is necessary to keep the message complexity low; as observed in [101], push or pull applied alone to contact the unmarked nodes can result in excessive message transmissions. Basically, push is more efficient initially when a large number of unmarked nodes need to be contacted, while pull works better toward the end when fewer unmarked nodes remain. It is also important that the scheme for the group construction ensures that none of the constructed groups are too large. Otherwise, a single red node can receive a large number of simultaneous calls during the Gossip phase resulting in an increased number of rounds.

## Description of the Algorithm

We now describe each of the four phases in greater detail. In each phase, communication between nodes takes place in rounds. Although, strictly speaking, each node is allowed to exchange information with only one partner in a given round, for convenience, in certain phases, we allow nodes to communicate with multiple nodes in a round. However, while calculating the total number of rounds for a phase, we count the multiple communications involving a single node as separate rounds. For instance, in Phase 2, a node can exchange information with $O(\log\log n)$ nodes (while dropping any extra requests) in a single round. Thus, even though Phase 2 has only $O(\log n)$ rounds, we compute the time complexity of Phase 2 as $O(\log n \log\log n)$ rounds.

In our analysis, we use several well-known results to bound the tail probability of a random variable (*e.g.*, Chernoff bounds, Azuma's inequality). These are included in Section 5.6 for easy reference.

**Phase 1** Push

```
1: Each node independently decides to remain active with probability $1/\log n$ or else becomes
   inactive.
2: Let $A$ be the set of all nodes that decide to remain active.
3: Each $u \in A$ marks itself *red*.
4: for $\frac{\log n \log\log n}{1-\delta}$ rounds do
5:     Each $u \in A$ selects a node $v$ independently and uniformly at random from the set of all
       nodes.
6:     for all $v$ that are unmarked do
7:         $v$ selects a node $u$ at random from the red nodes that contacted $v$.
8:         $u$ and $v$ exchange values and each stores the value $\max(\text{val}(u), \text{val}(v))$.
9:         $v$ points to $u$ and marks itself *blue*.
10:    end for
11: end for
12: All marked (*red* and *blue*) nodes become inactive.
```

**Lemma 1.** *The number of unmarked nodes at the end of Phase 1 is $\Theta(n/\log n)$ w.h.p.*

*Proof.* After step 1 of Phase 1, using standard Chernoff-bound type arguments, we have $n/\log n \pm \sqrt{n}$ *red* nodes w.h.p. Each of these *red* nodes tries to contact $\frac{\log n \log\log n}{1-\delta}$ nodes independently and

**Phase 2** Pull

```
1: Let $B \subset V$ denote the set of all *unmarked* nodes.
2: for $\frac{2\log n}{\log(1/8\delta)}$ rounds do
3:   Each $u \in B$ selects a node $v$ independently and uniformly at random from the set of all nodes.
4:   for all $v$ that are *unmarked* do
5:     Drop all requests.
6:   end for
7:   for all $v$ that are marked *blue* do
8:     Drop all but $\frac{4}{\delta(1-\delta)}\log\log n$ requests from nodes in $B$.
9:     for all $u \in B$ that contacted $v$, and are not dropped do
10:      $v$ sends $u$ a pointer to its *red* parent $w$.
11:      $u$ contacts $w$.
12:    end for
13:  end for
14:  for all $v$ that are marked *red* do
15:    Drop all but $\frac{4}{\delta(1-\delta)}\log\log n$ requests from nodes in $B$ (including the requests forwarded by the *blue* nodes in the group).
16:    for all $u \in B$ that contacted $v$, and are not dropped do
17:      $u$ points to $v$ and marks itself *blue*.
18:      $u$ and $v$ exchange values.
19:    end for
20:  end for
21: end for
```

uniformly at random (one connection attempt per round). Hence, the total number of connection attempts is $\frac{n\log\log n}{1-\delta}\left(1 \pm \frac{\log n}{\sqrt{n}}\right)$ w.h.p. Fix an unmarked node $u$. We now determine the probability that $u$ receives none of these messages.

At each step, this can happen because of two reasons: either the connection failed, or the connection succeeded, but went to a different node. Let $X_i$ be an indicator random variable (r.v.) such that $X_i$ is 1 if node $i$ was not contacted in Phase 1 by a red node and 0 otherwise. Then, the number of unmarked nodes at the end of Phase 1 is given by $X = \sum_{i=1}^{n} X_i$.

Let $c$ denote the total number of connection requests. As shown above, w.h.p., we have the following.

$$\frac{n\log\log n}{1-\delta}\left(1 - \frac{\log n}{\sqrt{n}}\right) \leq c \leq \frac{n\log\log n}{1-\delta}\left(1 + \frac{\log n}{\sqrt{n}}\right)$$

Now, the probability that $u$ is not contacted in any of the $c$ connection requests is given by the

**Phase 3** Gossip

```
1: Let $A$ be the set of all *red* nodes.
2: for $\left(\frac{3\log n}{(1-\delta)^2} + \log_{\frac{17}{16}} n\right)$ rounds do
3:   Each node in $A$ selects a node independently and uniformly at random from the set of all nodes.
4:   for all *blue* nodes ($v$) that are contacted do
5:     Drop all but $\frac{2}{\delta(1-\delta)}\log\log n$ requests from nodes.
6:     for all *red* nodes ($u$) whose requests have not been dropped do
7:       $v$ passes on its parent $w$'s address to $u$.
8:       $u$ then contacts $v$'s *red* parent $w$.
9:     end for
10:  end for
11:  for all *red* nodes ($v$) that are contacted do
12:    Drop all but $\frac{2}{\delta(1-\delta)}\log\log n$ requests from nodes.
13:    for all *red* nodes ($u$) whose requests have not been dropped do
14:      $u$ and $v$ compare values. The node with the smaller value replaces its value with the higher one.
15:    end for
16:  end for
17: end for
```

following.

$$\begin{aligned}
\Pr[X_i = 1] &= (\delta + (1-\delta)(1-1/n))^c \\
&\le \left(1 - \frac{1-\delta}{n}\right)^{\frac{n}{1-\delta}\log\log n\left(1-\frac{\log n}{\sqrt{n}}\right)} \\
&\le e^{-\log\log n\left(1-\frac{\log n}{\sqrt{n}}\right)} \\
&= (\log n)^{-\left(1-\frac{\log n}{\sqrt{n}}\right)} \\
&\le 2/\log n
\end{aligned}$$

Similarly, we obtain that

$$\Pr[X_i = 1] \ge \left(1 - \frac{1-\delta}{n}\right)^{\frac{n}{1-\delta}\log\log n\left(1+\frac{\log n}{\sqrt{n}}\right)}$$

Since $e^t(1-t^2/n) \le (1+t/n)^n$, we have

$$\begin{aligned}
\Pr[X_i = 1] &\ge e^{-\log\log n\left(1+\frac{\log n}{\sqrt{n}}\right)}\left(1 - \frac{2(1-\delta)\log\log n}{n}\right) \\
&= \log n^{-\left(1+\frac{\log n}{\sqrt{n}}\right)}\left(1 - \frac{2(1-\delta)\log\log n}{n}\right) \\
&\ge \frac{1}{4\log n}
\end{aligned}$$

Hence, it follows that $E[X] \in \Theta(n/\log n)$ and applying Azuma's inequality, we have

$$\Pr[|X - E[X]| > \epsilon E[X]] \le 2e^{-\frac{\epsilon^2 n^2}{2n\log^2 n}}$$

Finally, using Lemma 4 (in Section 5.6) and setting $\epsilon = \frac{\log^{3/2} n}{\sqrt{n}}$, it follows that $\Pr[X > (1+\epsilon)2n/\log n] \leq 1/n^{\Omega(1)}$ and $\Pr[X < (1-\epsilon)n/4\log n] \leq 1/n^{\Omega(1)}$. Because $0 < \epsilon < \frac{1}{2}$, it follows that $\frac{n}{8\log n} < X < \frac{4n}{\log n}$ w.h.p. □

**Corollary 1.** *The number of blue nodes at the end of Phase 1 is $\Theta(n - n/\log n)$ w.h.p.*

*Proof.* Follows directly from Lemma 1 by noting that the number of red nodes is $\Theta(n/\log n)$ w.h.p. □

**Lemma 2.** *Every node that was unmarked at the end of Phase 1 attaches itself to a red node by the end of Phase 2 w.h.p.*

*Proof.* Consider the event that an unmarked node fails to attach to a red node at the end of its $2\log n/\log(1/8\delta)$ calls. An unmarked node's call fails to attach it to a red node *if and only if* one of the following five bad events happen.

1. The call fails. This event occurs with probability $\delta$.
2. The call succeeds but lands in another unmarked node. By Lemma 1, the number of unmarked nodes is at most $4n/\log n$. Therefore, the probability of this event is at most $4(1-\delta)/\log n$.
3. The call succeeds and lands in a blue node $v$. However, the subsequent call to attach to $v$'s red parent fails. By Corollary 1, the number of blue nodes is at most $n - n/8\log n$. Hence, the probability of this bad event is at most $\delta\,(1-\delta)\left(1 - \frac{1}{8\log n}\right)$.
4. The call lands in a red node that has already received $\frac{4}{\delta(1-\delta)}\log\log n$ connections, and is dropped as a result. The probability of this event is at most the probability that a red node receives more than $\frac{4}{\delta(1-\delta)}\log\log n$ connections. The latter can be bounded from above as follows. Consider any red node $r$, and let $g$ denote its group size. Then, for every round of the pull phase, because at most $\frac{4n}{\log n}$ unmarked nodes participate, the probability that $r$ receives more than $\frac{4}{\delta(1-\delta)}\log\log n$ is equivalent to the probability of getting $\frac{4}{\delta(1-\delta)}\log\log n$ successes in $\frac{4n}{\log n}$ binomial trials, each with success probability at most $\frac{g(1-\delta)^2}{n}$. The expected

number of successes is at most $\frac{4g(1-\delta)^2}{\log n} < 4(1-\delta)\log\log n$. From Markov's inequality, the probability that the number of connections $Y$ is greater than $\frac{4}{\delta(1-\delta)}\log\log n$ equals $\Pr[Y > \frac{4}{\delta(1-\delta)}\log\log n] \leq \delta(1-\delta)^2 \leq \delta$.

5. The call lands in a blue node that has already received $\frac{4}{\delta(1-\delta)}\log\log n$ connections, and is dropped as a result. This is similar to the previous case (for red nodes) but with success probability at most $\frac{1-\delta}{n}$. Thus, the probability of this bad event is at most $\delta(1-\delta)^2 \leq \delta$.

Putting all this together and observing that we have $\delta$ as some constant greater than $1/\log n$ and less than $1/8$, we obtain the following:

$$\begin{aligned}
&\Pr[\text{Node fails to attach}]\\
&\leq \left(\delta + \tfrac{4(1-\delta)}{\log n} + \delta\left(1-\delta\right)\left(1 - \tfrac{1}{8\log n}\right) + 2\delta\right)^{\frac{2\log n}{\log(1/8\delta)}}\\
&\leq (\delta + 4\delta(1-\delta) + \delta(1-\delta) + 2\delta)^{2\log n/\log(1/8\delta)}\\
&\leq (8\delta)^{2\log n/\log(1/8\delta)}\\
&= \frac{1}{n^2}
\end{aligned}$$

Applying a union bound over all the unmarked nodes, it follows that the probability of a node remaining unmarked is at most $\frac{1}{n\log n}$. In other words, every node that was unmarked at the end of Phase 1 attaches itself to a *red* node by the end of Phase 2 w.h.p. □

Remark At the end of Phase 2, all the group sizes are $O(\log n \log\log n)$. This follows because the first phase runs for $O(\log n \log\log n)$ rounds, and in each round, a red node contacts at most one node. Further, the second phase runs for $O(\log n)$ rounds, and in each round, a red node contacts at most $O(\log\log n)$ nodes.

**Lemma 3.** *At the end of Phase 3, at least* $\Omega\left(\frac{(1-\delta)n}{\log n \log\log n}\right)$ *nodes have the max value w.h.p.*

*Proof.* A call is now equivalent to at most two successive calls (if the call lands in a blue node, then we have to make another call to connect to its red parent). Define the new call failure probability as $\delta^{'} = 1-(1-\delta)^2$. Let $\phi_t$ be the number of *red* nodes with the max value at the end of round $t$ of Phase 3. Let $\phi_0$ denote the number of red nodes with the MAX before the start of Phase 3.

We first prove that $\phi_\tau > (1-\sqrt{2/3})\log n$ after $\tau = \frac{3\log n}{1-\delta'}$ rounds. If $\phi_0 > \log n$, we are done. Consider the case when $\phi_0 < \log n$. Since $\phi_0 < \log n$, if a call does not fail, then it will contact a new node w.h.p. Therefore we are only interested in finding the number of rounds before $\log n$ successful calls are made. Define $X_i = 1$ as the probability that call $i$ from the max node succeeds. Let $X = \sum_{i=1}^{(3\log n)/1-\delta'} X_i$ be the total number of successful calls in $\frac{3\log n}{1-\delta'}$ rounds. The probability that a call succeeds is $\Pr[X_i = 1] = 1-\delta'$. Hence, $E[X] = 3\log n$.

Applying Azuma's inequality and setting $\epsilon = \sqrt{2/3}$:

$$\begin{aligned}\Pr[|X - E[X]| > \epsilon E[X]] &< 2\exp\left(-\frac{\epsilon^2 E[X]^2}{2\log n}\right)\\ &= 2\exp\left(-\frac{9\epsilon^2 \log n}{6}\right)\\ &= 2\exp\left(-\log n\right)\\ &= 2/n\end{aligned}$$

Thus, w.h.p., at the end of $\tau = \frac{3\log n}{1-\delta'}$ rounds, we have $\phi_\tau > \left(1-\sqrt{2/3}\right)\log n$. Once this happens, as we show below, we enter an exponential growth phase and in the next $O(\log n)$ rounds, about $O(\frac{n}{\log n \log\log n})$ red nodes will have the MAX.

Let us re-number the rounds to simplify the expressions. Let us number the first round with at least $(1-\sqrt{2/3})\log n$ red nodes having the MAX as round 0 (note that such a round exists based on previous arguments). Now let us compute the value of $\phi_{t+1}$ given that in the current round $\phi_t$ red nodes have the MAX. We have $\phi_0 > \left(1-\sqrt{2/3}\right)\log n$. Since we have $\phi_t$ red nodes with MAX, there will be at least $\phi_t$ messages containing MAX in the current round (ignoring pull transmissions). In the rest of this proof, we only consider these messages containing MAX.

Let $X_i$ be the indicator r.v. that denotes whether message $i$ (containing MAX) is successful or not: a message is successful if the call succeeds and the MAX reaches a node that has not already been informed about the MAX. Then, $X = \sum_{i=1}^{\phi_t} X_i$ is the number of successful messages. We also have $1-\delta' = (1-\delta)^2$.

A message can fail *if and only if* one of the following happens.

1. Either the first or the second call fails. This event occurs with probability $\delta'$.

2. Recall that some connections are dropped if a node receives more than $\frac{2}{\delta(1-\delta)} \log\log n$ connections. The probability of this event at the first node (a blue node) is at most $\delta(1-\delta)$ (the proof is very similar to the proof in Lemma 2 for the case where a blue node receives more than $\Omega(\log\log n)$ calls), while the probability of this event at the second contacted node (red node that is the parent of the blue node in the first call) is at most $\delta(1-\delta') < \delta(1-\delta)$. Thus, the total probability is at most $2\delta(1-\delta)$.

3. Both calls succeed, but the group contacted already contains the MAX. The probability of this event occurring is at most $\frac{(1-\delta')\phi_t \log n \log\log n}{n}$, because the maximum group size of a red node is $\frac{\log n \log\log n}{1-\delta}$ (by construction) and there are at most $\phi_t$ red nodes with MAX.

4. Both calls succeed, but the group contacted is simultaneously contacted by another node with the MAX. In this case, we conservatively assume that this message is wasted; in other words, if two messages reach the same node, then both messages are considered unsuccessful. This event occurs with probability at most $\frac{(1-\delta')\phi_t \log n \log\log n}{n}$.

Thus, the probability that a message is wasted can be upper-bounded as follows.

$$\Pr[X_i = 0] \quad \leq \quad \delta' + 2\delta(1-\delta) + \frac{2(1-\delta')\phi_t \log n \log\log n}{n}$$

Since we have $\phi_t < \frac{n}{16 \log n \log\log n}$,

$$\begin{aligned}
\Pr[X_i = 0] &\leq 1-(1-\delta)^2 + 2\delta(1-\delta) + \frac{(1-\delta)^2}{8} \\
&= 1 - \frac{7}{8}(1-\delta)^2 + 2\delta(1-\delta) \\
&\leq \frac{1}{8} + \frac{15\delta}{4} \\
\Pr[X_i = 1] &\geq 1 - \left(\frac{1}{8} + \frac{15\delta}{4}\right) \\
&= \frac{7}{8} - \frac{15\delta}{4} \\
E[X] &\geq \left(\frac{7}{8} - \frac{15\delta}{4}\right)\phi_t
\end{aligned}$$

Since $\delta < 1/8$, the above expression implies that, in expectation, the number of nodes with MAX grows exponentially from one round to the next. To make this claim w.h.p., we apply Azuma's inequality to show a sharp concentration result for the expected number of successful messages.

$$\Pr[|X - E[X]| > \epsilon E[X]] \quad < \quad 2\exp\left(-\frac{\epsilon^2(E[X])^2}{2\phi_t}\right)$$

Since $E[X] \geq \left(\frac{7}{8} - \frac{15\delta}{4}\right)\phi_t$, we have

$$\Pr[|X - E[X]| > \epsilon E[X]] < 2\exp\left(-\frac{\epsilon^2}{2}\left(\frac{7}{8} - \frac{15\delta}{4}\right)^2\phi_t\right)$$

Since $\phi_t > \left(1 - \sqrt{2/3}\right)\log n$ and $\left(\frac{7}{8} - \frac{15\delta}{4}\right)^2 > 0$, setting $\epsilon = 1/2$, we obtain

$$\Pr\left[X < \frac{1}{2}\left(\frac{7}{8} - \frac{15\delta}{4}\right)\phi_t\right] \quad < \quad \frac{2}{n^{O(1)}}$$

Hence, w.h.p., the number of red nodes with MAX in round $t+1$ is

$$\begin{aligned} \phi_{t+1} \quad &> \quad \phi_t + \frac{1}{2}\left(\frac{7}{8} - \frac{15\delta}{4}\right)\phi_t \\ &= \quad \frac{23 - 30\delta}{16}\phi_t \\ &> \quad \frac{17}{16}\phi_t \end{aligned}$$

The last step above follows since $\delta < 1/8$. Thus, w.h.p., after $\left(\frac{3\log n}{1-\delta'} + \log_{\frac{17}{16}} n\right)$ rounds, at least $\Omega\left(\frac{(1-\delta)n}{\log n \log\log n}\right)$ red nodes will have the MAX. □

**Theorem 1.** *Any node can compute the value of MAX using* $O(n\log\log n)$ *messages and* $O(\log n \log\log n)$ *rounds of communication.*

*Proof.* From Lemma 3, it follows that at least $\frac{cn}{\log n \log\log n}$ red nodes (for some $c > 0$) have the MAX. Once Phases 1, 2 and 3 are complete, any node that is interested in MAX requests $\log n$ other nodes to "sample" the MAX. Each of these $\log n$ nodes then successfully samples $\frac{2}{c}\log n \log\log n$ nodes, and returns the maximum observed value to the querying node. The probability that none of the nodes with the MAX is sampled is at most $(1 - \frac{c}{\log n \log\log n})^{\frac{2}{c}\log^2 n \log\log n} < \frac{1}{n^2}$. Thus, the maximum of all the values obtained from the delegated nodes is the actual maximum w.h.p.

To bound the total number of communication rounds, observe that the number of communication rounds in each phase is $O(\log n \log\log n)$. This can be easily seen from the description of each phase.

To bound the total number of messages, we note that the number of messages in Phase 1 is $O(n \log \log n)$. By Lemma 1, the number of unmarked nodes that take part in Phase 2 is $\Theta(n/\log n)$, and hence the message complexity of Phase 2 is $O(n)$. The message complexity of Phase 3 is $O(n)$ because the number of red nodes is $\Theta(n/\log n)$, and each red node makes $O(\log n)$ calls. The message complexity of sampling is simply the number of samples that need to be drawn and is therefore $O\left(\log^2 n \log \log n\right)$. Thus, the overall message complexity is $O(n \log \log n)$. □

**Corollary 1.** *All nodes can compute the value of MAX using $O(n \log \log n)$ messages and $O(\log n \log \log n)$ rounds of communication.*

*Proof.* If all nodes want MAX, then we first perform Phases 1, 2 and 3 as before. At the end of Phase 3, each node decides to be a propagator of MAX with probability $2 \log n/n$. It can be shown that there will be $\Theta(\log n)$ propagators w.h.p. Next, each propagator gets MAX by sampling $\log n$ other nodes as described in the first paragraph of the proof of Theorem 1. After this, applying the result from [101], the propagator nodes can disseminate MAX to $\Omega((1-\delta)n)$ nodes using $O(n \log \log n)$ messages and $O(\log n)$ rounds.

After this, we can form groups as in Phases 1, 2, but with red nodes chosen from the nodes that already have MAX. As every node belongs to some group, every node also has MAX. Essentially, Phase 1 is performed with a minor modification. Only nodes who have MAX perform step 1 of Phase 1. Since $\delta$ is a constant and at least $\Omega((1-\delta)n)$ nodes have MAX, we still have $\Theta(n/\log n)$ red nodes at the end of this step. The rest of the Phase 1 proceeds as before. Phase 2 is unchanged. Since at the end of these two phases, each node has successfully communicated with a red node, all nodes have MAX. Phases 1 and 2 together take $O(\log n \log \log n)$ rounds and $O(n \log \log n)$ messages as seen above. □

### 5.4.3 Computation of SUM, AVERAGE, RANK

We now extend our MAX computation scheme to estimate the sum and average of node values. The key idea behind our algorithm is the following. First, groups of nodes are formed using Phases 1, 2 of the MAX computation algorithm. As before, the group heads will be referred

**Algorithm 4** Modified-Push-Sum($x_{r_1} \dots x_{r_m}$)

```
1: $s_{u,t}$ is node $u$'s value in round $t$. $s_{u,0} = x_u$ for each $u \in A$.
2: $w_{u,t}$ is node $u$'s weight in round $t$. $w_{u,0} = 1$ for each $u \in A$.
3: for $O(\log m + \log(1/\epsilon))$ rounds do
4:     Each node $u \in A$ independently and uniformly at random calls a node $v \in \{1 \dots n\}$. If $v$ is
       a blue node, $u$ is directed to the group head of $v$ in the subsequent round. Note that the
       group head is a red node from $A$.
5:     Let $Y_{v,t}$ be the set of nodes that called $v$ in round $t$.
6:     $s_{v,t} = s_{v,t-1}/2 + \sum_{u \in Y_{v,t}} s_{u,t-1}/2$.
7:     $w_{v,t} = w_{v,t-1}/2 + \sum_{u \in Y_{v,t}} w_{u,t-1}/2$.
8:     The current estimate of the average at node $v$ is $s_{v,t}/w_{v,t}$.
9: end for
```

**Algorithm 5** Compute-Average

```
1: Form groups as in Phases 1, 2 of MAX computation. Let $A = \{r_1, \dots, r_m\}$ be the set of red
   nodes, and for each red node $u \in A$, let $g_u$ denote its group size (that is, the size of red node
   $u$ and all the blue nodes attached to it).
2: Use the MAX finding scheme presented earlier to find the maximum size group for all the red
   nodes. Since all red nodes can find MAX, red node $r$ can determine that it has the maximum
   sized group (break ties using node ids in the messages used for MAX computation).
3: Let $y_u = \sum_{v \in \mathrm{group(u)}} val(v)$.
4: All the red nodes compute $g_{\mathrm{avg}}$ using Modified-Push-Sum($g_{r_1}, \dots, g_{r_m}$).
5: All the red nodes compute $y_{\mathrm{avg}}$ using Modified-Push-Sum($y_{r_1}, \dots, y_{r_m}$).
6: Node $r$ computes the average $\hat{\mu} = y_{avg}/g_{avg}$, and communicates it to all nodes using the
   rumor-spreading scheme in [101].
```

to as the *red* nodes and the other nodes will be referred to as the *blue* nodes. During group formation, every *red* node maintains the size of its group, and the sum of the values within its group. Next, the group heads use the MAX computation algorithm to compute the maximum group size (for reasons that will become clear soon). Finally, the *red* nodes use the gossip-based Push-Sum algorithm in [107] to compute the average of node values. As we will show later, owing to the distinct group sizes, one can only ensure that the true-average resides in the *red* node with the largest group size. As the nodes already compute the largest group size, the *red* node with the largest group knows its identity and hence also knows that the value it has at the end of the protocol is the true-average (with a small relative error).

Algorithm 4 is a modified version of the Push-Sum protocol in [107]. Let $A = \{r_1 \dots r_m\}$ be the set of red nodes after Phases 1, 2 of MAX computation. In our version, the Push-Sum protocol computes the average of the set of values $x_{r_1}, \dots, x_{r_m}$ for only the red nodes. Note that every other node will be child of one of the red nodes, and in the modified protocol, any call to these nodes will be forwarded to its parent in $A$.

Let $\alpha$ denote the true average of the $x_{r_i}$'s. Further, let $r$ be the red node with the maximum group size and $x_{avg}$ denote the average computed at node $r$.

**Theorem 2.** *At the end of Algorithm 4, $|x_{avg} - \alpha|/\alpha \leq \varepsilon$ for any $\varepsilon > 0$. Furthermore, the total number of messages is $O(m(\log m + \log \frac{1}{\varepsilon}))$ and the number of rounds is $O(\log m + \log \frac{1}{\varepsilon})$.*

The proof of Theorem 2 is along similar lines as the proof in [107]. We will need some definitions as in [107] followed by a couple of key lemmas. Let $\vec{v}_{u,t}$ be a vector for node $u$ in round $t$, of which the $z^{\text{th}}$ component $v_{u,z,t}$ denotes the fraction of node $z$'s value that is currently part of $u$'s sum $s_{u,t}$ in round $t$. Thus, the sum at node $u$ in round $t$, $s_{u,t} = \sum_z v_{u,z,t} x_z$. As shown in [107], the invariant $\sum_u v_{u,z,t} = 1 \ \forall z$ holds for Algorithm 4 as well. Similar to [107], define the following potential function for round $t$.

$$\Phi_t = \sum_{u,z} (v_{u,z,t} - \tfrac{w_{u,t}}{m})^2$$

Above, $m = \Theta(n/\log(n))$ is the number of *red* nodes, and $w_{u,t} = \sum_z v_{u,z,t}$. We now state the following lemma which is a variant of a similar result proved in [107].

**Lemma 5.4.1.** *The following holds:*

$$E[\Phi_{t+1} | \Phi_t = \phi] = \tfrac{1}{2}(1 - \textstyle\sum_{u \in A} p_u^2)\phi \ , \tag{5.1}$$

*where $p_u = (1-\delta)^2 g_u / \sum_{v \in A} g_v$.*

*Proof.* This proof is very similar to the proof in [107]. The only difference is that $p_u$, the probability with which a node $u$ is called in a round, is no longer uniform, but is skewed based on the group size $g_u$. □

We also need the following lemma which is somewhat different from [107].

**Lemma 5.4.2.** *There exists a $\tau \in O(\log m)$ such that after $\tau' > \tau$ rounds of execution of Modified-Push-Sum, $w_{r,\tau'} \geq 2^{-\tau}$ w.h.p.*

*Proof.* In [107], the above result is proved for all nodes and not just the node $r$ with the maximum group size. The proof in [107] relies on the fact that when every node is contacted with uniform probability, each node receives a fraction of every other node's value after $\tau$ rounds. In our case, since the distribution is skewed, we cannot guarantee the lower bound on weight for each and every node. However, it is easy to see that the red node with the largest group size has a higher probability of being contacted. Thus, we can show that it receives a fraction of every other node's value after $\tau$ rounds, and thus its weight satisfies the lower bound. □

*of Theorem 2.* The proof is along the lines of [107], and employs the results of Lemmas 5.4.1 and 5.4.2. Due to Lemma 5.4.1, we get $\Phi_t \leq m2^{-t}$ (intuitively, $\Phi_t$ decreases by a constant factor in each round and is $m$ when $t = 0$). By choosing $t = \log m + 2\log\frac{1}{\varepsilon} + 2\tau$ ($\tau$ as in lemma 5.4.2), we can show that $\Phi_t \leq \varepsilon^2 2^{-2\tau}$ for any $\varepsilon > 0$. This implies that $|v_{r,z,t} - \frac{w_{r,t}}{m}| \leq \varepsilon 2^{-\tau}$ for all $z$, or in other words $|\frac{v_{r,z,t}}{w_{r,t}} - \frac{1}{m}| \leq \frac{\varepsilon 2^{-\tau}}{w_{r,t}}$. Lemma 5.4.2 gives us the required lower bound of $w_{r,t} \geq 2^{-\tau}$ to have $|\frac{v_{r,z,t}}{w_{r,t}} - \frac{1}{m}| \leq \varepsilon$. Note that $\frac{v_{r,z,t}}{w_{r,t}}$ represents the contribution of $z$'s value at $r$ and w.h.p. this is approximately $\frac{1}{m}$ for all nodes. This implies that after $t \in O(\log m + \log\frac{1}{\varepsilon})$ rounds, the average $x_{avg}$ computed at $r$ has relative error at most $\epsilon$. □

Algorithm 5 uses our Modified-Push-Sum procedure to compute $y_{avg}$ and $g_{avg}$, the average group value and average group size, respectively. Let $\hat{\mu} = \frac{y_{avg}}{g_{avg}}$ be the estimate of average as computed by the red node $r$ with the largest group, and let $\mu$ be the actual average of all the values $val(u)$ across nodes that did not fail at the beginning.

Our main result of this subsection is the following.

**Theorem 3.** *At the end of Algorithm 5, $|\hat{\mu} - \mu|/\mu \leq 3\epsilon$ for any $\epsilon > 0$. Furthermore, the total number of messages is $O(n(\log\log n + \log\frac{1}{\epsilon}))$ and the number of rounds is $O(\log n \log\log n + \log\frac{1}{\epsilon})$.*

*Proof.* It is easy to see that $y_{\text{avg}}$ and $g_{\text{avg}}$ are computed with relative error at most $\epsilon$ at node $r$ (follows from Theorem 2). This implies that the relative error in the computation of $\hat{\mu} = y_{\text{avg}}/g_{\text{avg}}$ is at most $3\epsilon$. The message complexity of Modified-Push-Sum among $m = \Theta(n/\log n)$ *red* nodes is simply $O(m(\log m + \log\frac{1}{\varepsilon})) = O(n + n\log\frac{1}{\varepsilon})$ and the time complexity is $O(\log m + \log\frac{1}{\varepsilon}) = O(\log n + \log\frac{1}{\varepsilon})$. Thus the message and time complexity are dominated by the formation of groups, and are $O(n\log\log n)$ and $O(\log n \log\log n)$, respectively. □

**Corollary 2.** *All nodes can find the true average of node values using $O(\log n \log\log n)$ rounds of communication and $O(n \log\log n)$ total messages.*

Given the average, the sum can be computed by just multiplying the average group value $y_{avg}$ by the number of groups (which can be estimated using Modified-Push-Sum with only $x_r = 1$ and the remaining $x_i$s equal to 0). Using this algorithm for computing the sum, computing the rank of a given value is also straightforward. The value $x$ whose rank needs to be computed can be disseminated to all the nodes, and every node now keeps a new value which is 1 if its original value is less than $x$ and 0 otherwise. Computing the sum of these new values gives the rank of $x$. Since dissemination and sum computation can be done in $O(\log n \log\log n)$ rounds and $O(n \log\log n)$ messages, we have the following corollary.

**Corollary 3.** *The rank of any value (the value is ranked among the values for nodes that did not fail) can be computed using $O(\log n \log\log n)$ rounds of communication and $O(n \log\log n)$ total messages.*

## 5.5 Conclusion

In this chapter, we presented a novel gossip-based scheme for computing common aggregates like MIN, MAX, SUM, AVERAGE and RANK of node values using $O(n \log\log n)$ messages and in $O(\log n \log\log n)$ rounds of communication. To the best of our knowledge, this is the first result to show that these aggregates can be computed with high probability using only $O(n \log\log n)$ messages. Thus, compared to the previously best known results for distributed aggregate computation by Kempe et al., our scheme significantly reduces the communication overhead (a factor of $O(\log n/\log\log n)$) while causing only a modest increase (a factor of $O(\log\log n)$) in the number of rounds.

We conjecture that our results achieve the lower bound for message complexity in the gossip model since previous work by Karp et al. showed that even simpler problems like rumor dissemination require at least $\Omega(n \log\log n)$ messages regardless of the number of rounds. Formally deriving the lower bounds for message and time complexity for aggregate computation using gossip-style communication remains a topic for future work.

## 5.6 Technical Desiderata

We present some of the existing results in probability that we use extensively in our proofs. We use the following inequality to apply the method of bounded differences.

**Theorem 4** (Azuma's Inequality [125]). *Let $Y_0, Y_1, \ldots$ be a martingale sequence such that for each $k$,*

$$|Y_k - Y_{k-1}| \leq c_k$$

*where $c_k$ may depend on $k$. Then for all $t \geq 0$ and any $\lambda > 0$,*

$$\Pr[|Y_t - Y_0| \geq \lambda] \leq 2exp\left(-\frac{\lambda^2}{2\sum_{k=1}^{t} c_k^2}\right)$$

Since the method of bounded differences is applied frequently, we briefly revisit it here. The following content is from [125]. Let $X_1, \ldots, X_n$ be *any* sequence of random variables. Let $f(X_1, \ldots, X_n)$ be some function defined over these random variables. The function $f$ is said to satisfy the Lipschitz condition if an arbitrary change in the value of any one argument of the function does not change the value of the function by more than 1. The sequence of random variables $Y_0 = E[f(X_1, \ldots, X_n)]$, $Y_i = E[f(X_1, \ldots, X_n)|X_1, \ldots, X_i]$ and $Y_n = f(X_1, \ldots, X_n)$ forms a martingale sequence. If $f$ is Lipschitz, then for $1 \leq i \leq n$, $|Y_i - Y_{i+1}| \leq 1$. This condition is clearly satisfied for the sum of indicator random variables. We can then use Azuma's inequality to bound the probability that a sum of indicator random variables deviates from the expected value of that sum.

**Theorem 5** (Chernoff bounds [125]). *Let $X$ be a sum of independent and identically distributed 0/1 random variables. Let $\mu$ denote the expected value of $X$. Then we have:*

1. $\Pr[X \leq (1-\epsilon)\mu] \leq exp\left(-\frac{\mu\epsilon^2}{2}\right)$ *for all* $0 < \epsilon < 1$.

2. $\Pr[X \geq (1+\epsilon)\mu] \leq exp\left(-\frac{\mu\epsilon^2}{3}\right)$ *for all* $0 < \epsilon < 1$.

3. $\Pr[X \geq (1+\epsilon)\mu] \leq exp\left(-\frac{\mu\epsilon^2}{2+\epsilon}\right)$ *for all* $\epsilon > 1$.

From the statement above, we can infer that $\Pr[X \geq (1+\epsilon)\mu] \leq \exp\left(-\frac{\mu\epsilon}{2}\right)$ for all $\epsilon \geq 2$.

We also use the following simple fact frequently in the proofs.

**Lemma 4.** *If $\mu_1 \leq \mu \leq \mu_2$, then:*

1. $\Pr[X > (1+\epsilon)\mu_2] \leq \Pr[X > (1+\epsilon)\mu] \leq \Pr[X > (1+\epsilon)\mu_1]$.

2. $\Pr[X < (1-\epsilon)\mu_1] \leq \Pr[X < (1-\epsilon)\mu] \leq \Pr[X < (1-\epsilon)\mu_2]$.

*Proof.* For any two events $A$ and $B$ if $A \Rightarrow B$, we have $\Pr[A] \leq \Pr[B]$:

$$\begin{aligned} Pr[\text{A and B}] = \Pr[B|A]\Pr[A] &= \Pr[A|B]\Pr[B] \\ \Pr[A] &= \Pr[A|B]\Pr[B] \\ \Pr[A] &\leq \Pr[B] \end{aligned}$$

The lemma follows since $(X > (1+\epsilon)\mu_2) \Rightarrow (X > (1+\epsilon)\mu)$ and $(X > (1+\epsilon)\mu) \Rightarrow (X > (1+\epsilon)\mu_1)$. Similarly, we have $(X < (1-\epsilon)\mu_1) \Rightarrow (X < (1-\epsilon)\mu)$ and $(X < (1-\epsilon)\mu) \Rightarrow (X < (1-\epsilon)\mu_2)$. □

# Chapter 6

# Data Monitoring (Continuous Queries)

This is joint work with Jeyashankher Ramamirtham, Rajeev Rastogi and Pushpraj Shukla. These results also appeared in [103].

## 6.1 Introduction

Monitoring emerging large-scale, distributed systems (like peer to peer systems, server clusters, IP networks and sensor networks) poses several interesting challenges. Sensor networks place various constraints on the communication and processing capabilities of the nodes. Network monitoring systems need to process large volumes of data in (near) real-time from a widely distributed set of sources, and it is nearly impossible to store and process the entire information in an on-demand fashion. Further, many queries in these systems are continuous and are executed for the lifetime of the system. For example, consider a system that monitors a large network for distributed denial of service (DDoS) attacks. This system needs to process data from several routers at a rate of several gigabits per second. Also, the system needs to detect attacks as soon as they happen (with minimal latency) to enable networks operators to take expedient countermeasures in order to mitigate the effect of these attacks. The research community has looked at developing algorithms for computing and tracking a wide range of aggregate statistics over distributed data streams [16, 39]. These apply to a general class of continuous monitoring applications, where the goal is to optimize the operational resource usage of these algorithms and still guarantee that the estimate of the aggregate function is always within specified error bounds.

Communication efficiency is a critical concern for these distributed data management systems. In sensor networks, transmitting messages from sensor nodes consumes valuable battery resources that determine the lifetime of these networks. In network data monitoring systems, each node in the network receives voluminous amounts of data. Transmitting an equivalent amount of data across the network in order to perform distributed computations is impractical. Thus, the communication efficiency of the distributed computation algorithms determines the practicality of these systems. Previous methods [91, 158, 51, 87] describe distributed constraints monitoring or distributed triggers as a mechanism of reducing the amount of communication. These methods filter out "uninteresting" events and do not require communication across the network for these events; thus, reducing the communication needed to perform the computations.

In this chapter, we introduce a new set of methods that we call *non-zero slack* schemes. We study these methods for an important class of queries called distributed SUM constraints or distributed triggers, that are used to track anomalous behavior. We quantify the benefits of non-zero slack schemes for distributed SUM constraints monitoring, both analytically and empirically, and show that these methods can considerably reduce the number of communication messages in the network (by 60% in our experiments). Implementing non-zero slack schemes for the simple distributed SUM constraint queries presents a number of challenges and we address these challenges in this chapter. We believe that non-zero slack schemes can be applied to a wide range of other distributed queries to make them communication efficient and thus, practical. We leave the study of non-zero schemes for other queries as future work.

**Distributed SUM constraints:** The distributed constraints that we focus on are of the form $\sum_{i=1}^{n} x_i \leq T$, where $n$ is the number of nodes in the system, $x_i$ is the value of a variable that is being monitored at node $i$, and $T$ is the constraint's threshold. This constraint can be decomposed into a set of local thresholds, $T_i$ at each remote site $i$, such that $\sum_{i=1}^{n} T_i \leq T$. If at all sites, $x_i \leq T_i$ then $\sum_{i=1}^{n} x_i \leq \sum_{i=1}^{n} T_i \leq T$. Thus, if none of the thresholds at the nodes are violated, the global constraint is satisfied. In effect, the local thresholds act as filters that help in reducing the amount of communication messages in the system.

Consider an example application for detecting service quality degradations of VoIP sessions in a network. Let us suppose that VoIP requires the end-to-end delay to be within 200 milliseconds and the loss probability to be within 1%. Consider a path through the network with $n$ network elements (routers, switches) $s_1, s_2, \ldots, s_n$. To monitor loss probabilities through the network, each network element has an estimate of its local loss probability, say $l_i, i \in [1, n]$. The loss probability of the path through these network elements is given by $L = 1 - (1 - l_1)(1 - l_2) \dots (1 - l_n)$. This yields $\log(1 - L) = \log(1 - l_1) + \log(1 - l_2) + \cdots + \log(1 - l_n)$. If we need $L \leq 0.01$, we need $\log(1 - L) \geq \log(0.99)$. Inverting the sign on both sides, this transforms into the constraint $\sum_{i=1}^{n} (-\log(1 - l_i)) \leq -\log(0.99)$.

Thus, the problem of monitoring losses in a network can be addressed using distributed constraints monitoring. Delays can be monitored similarly using distributed SUM constraints.

**Non-zero slack schemes**: Algorithms to determine local constraints can be classified into two categories: zero slack schemes and non-zero slack schemes.

- Zero slack schemes: These algorithms assign local constraints that do not have any *slack* in the system. Specifically, the local threshold values, $T_i$ are determined such that $\sum_{i=1}^{n} T_i = T$. Most prior work uses this method of *tight* allocation of threshold values at the nodes.

- Non-zero slack schemes: These algorithms determine local constraints that retain some slack in the system; i.e. $\sum_{i=1}^{n} T_i \leq T$ and the slack in the system is given by $S = T - \sum_{i=1}^{n} T_i$.

Zero slack schemes perform well if the values at the nodes do not exceed their threshold values, i.e. $x_i \leq T_i$. However, this is seldom the case and the values can exceed this value frequently even when the global sum is less than $T$. When a node's value exceeds this threshold value, we can no longer say with certainty if the global constraint is satisfied *or not*. This requires polling the values at all nodes in the network to determine whether the global constraint is still satisfied, causing a flurry of communication messages.

Non-zero slack schemes, on the other hand, retain some slack, and thus allow nodes to exceed their local thresholds by that amount and are still able to guarantee satisfaction of the global constraint without polling for values at all nodes. We demonstrate that the reduction in the messages is considerable both analytically and through experiments for typical data.

**Our contributions:** In this chapter, we undertake a comprehensive study of communication efficient monitoring of distributed SUM constraints using non-zero slack schemes.

1. We show both analytically and empirically that non-zero slack schemes outperform the state-of-the-art zero slack scheme for different data distributions.

2. We develop adaptive algorithms for setting threshold values at remote sites in the presence of non-zero slack (for changing data distributions).

3. Finally, we present the results of a thorough and detailed set of experiments using both synthetically generated data and real world data, and show that our adaptive non-zero slack algorithms can result in significant savings in the amount of communication.

To the best of our knowledge, our's is the first work to systematically study non-zero slack schemes for detecting distributed constraint violations.

The rest of this chapter is organized as follows. In Section 6.2, we review related work in the area of distributed monitoring. We formally define the problem of threshold assignment for distributed monitoring in Section 6.3 and show that non-zero slack schemes can result in lower communication costs for a distribution pattern. In Section 6.4, we describe our adaptive algorithms to set local threshold values. We present an experimental evaluation of the performance of our algorithms in Section 6.5 and finally conclude in Section 6.6.

## 6.2 Related Work

Monitoring data streams in a distributed environment has been an important focus area of research in recent years. Algorithms have been proposed for continuous monitoring of top-k items [16], sums and counts [135], quantiles [39], joins and max values. These papers address problems that are different from the problem we address in this chapter. For instance, Olston *et al.* [135] tackle the problem of *continuously tracking* multiple SUM queries with different error bounds which is very different from our problem which aims to detect if the result of a single SUM query exceeds a given threshold.

Most recent work on the problem of distributed constraint monitoring propose zero-slack schemes and one prior work describes a non-zero slack scheme for implementing distributed constraint monitoring.

**Zero slack schemes:** Jain *et al.* [91] discuss the challenges in implementing distributed triggering mechanisms for network monitoring. They use a zero slack scheme that uses local constraints of $T/n$ to detect constraint violations.

The recent work of Sharfman *et al.* [158] represents the state of the art in detecting distributed constraint violations. For SUM constraints over variables $x_i$, their scheme reduces to a zero slack adaptive scheme that always maintains the invariant $\sum_i T_i = T$. Each time a local constraint $x_i \leq T_i$ is violated, the $x_i$ values are polled and the slack $S = T - \sum_i x_i$ is distributed among the sites, that is, each $T_i$ is set to $T_i + S/n$. We compare our algorithms against this scheme, which we also refer to as the "Geometric Scheme" in later sections.

Agrawal *et al.* [6] present a zero slack scheme that formulates the problem of selecting local constraints as an optimization problem whose objective function aims to minimize the probability of global polls given the individual frequency distributions of variables $x_i$.

Keralapura *et al.* [108] propose static and adaptive algorithms to monitor distributed SUM constraints. We do not study static methods in this chapter as they result in much more communication than adaptive ones. Their adaptive schemes use similar methods to the ones proposed in [158] and are essentially zero slack schemes.

**Non-zero slack schemes:** The scheme that is perhaps closest to our approach is that of Dilman and Raz [51]. In addition to the Simple-Value scheme that sets each local threshold $T_i$ to $T/n$, they also propose an *Improved Value* scheme in which local thresholds $T_i$'s (same for all $i$) are set to lower values than $T/n$, observing that the *Improved Value* scheme can outperform the simple value scheme. This translates to the improvement of non-zero slack schemes over zero slack schemes in our work. Their paper however does not address the following issues that we study in this chapter - (1) They do not show how to set local thresholds, which is critical for achieving good performance. (2) They do not show how to adapt local threshold values for changing data distributions.

Huang *et al.* [97] consider a novel variant of the instantaneous tracking problem where they track constraint violations that *persist over time.* They use queuing theory as an analytical tool to compute local threshold values that meet user-specified false alarm and missed detection rates. Their analysis makes two assumptions which may not be true in our setting: (1) All local threshold values are equal, and (2) Local site values follow a Normal $N(0, \sigma^2)$ distribution (e.g., network link delay values have been found to follow a Weibull distribution). Also, it is unclear if the computed local threshold settings in [97] optimize total communication costs - this is because false alarms correspond to global polls and are only one component of communication costs (the second being local alarms). Note that [97] allows missed detections, but we do not.

We propose a non-zero slack scheme called the reactive scheme that is inspired by the probabilistic scheme presented in [136]. However, our problem setting is significantly more complex. Please refer to the end of Section 6.4.4 for a detailed discussion of the differences between our work and theirs as well as novel contributions of our work in this context.

## 6.3 Problem Definition

### 6.3.1 System Architecture

We consider a distributed monitoring system consisting of $n$ remote sites $s_1, \ldots, s_n$ and a central coordinator site $s_0$. Each site $s_i$ observes a continuous stream of updates, which it records as a constantly changing value of its local variable $x_i$. $x_i$'s can take non-negative real values in the domain $[0, \infty)$. Negative values can be handled as well. We omit it for clarity of discussion. The remote sites can communicate with the coordinator to send or receive messages. We assume that this communication can happen without any loss or delay. In addition, time is assumed to be slotted and synchronized across sites. At the end of each time slot $t$, site $s_i$ observes value $x_i(t)$. If time is not important, we refer to the value at the site $i$ as simply $x_i$. We shall sometimes refer to a slot as a 'round' for the system. Note that the values $x_i$ can increase or decrease with time. This system architecture is in line with previous work [108].

### 6.3.2 Detection of distributed constraints violations

We are looking to devise schemes that can detect violation of global constraints defined over distributed system variables of the form $\sum_i^n x_i \leq T$. Each site $s_i$ is assigned a local threshold $T_i$ such that $\sum_i^n T_i \leq T$. A remote site sends a *local alarm* to the coordinator whenever $x_i > T_i$ and remains silent otherwise. If $\forall i \in [1, n], x_i \leq T_i$ and $\sum_i^n T_i \leq T$, then $\sum_i^n x_i \leq T$. Thus, if none of the sites report local alarms, the global constraint is not violated.

If some site violates the local constraint, then the global constraint could have been violated. Let us assume that site $j$ violates the local constraint and sends the value at the site $x_j$ to the coordinator. The coordinator now verifies if $x_j + \sum_{i \neq j} T_i \leq T$ is satisfied. If this condition is satisfied, then the global constraint is not violated. However, if the constraint is not satisfied, the coordinator polls the remote sites for their exact values to determine if the global constraint is still satisfied. We refer to this as a *global poll*.

A global poll can be performed by polling for the exact value at *all* sites. Alternatively, the coordinator site can poll a subset of sites $S$ and determine if the global constraint is not violated by checking if $\sum_{i \in S} x_i + \sum_{i \notin S} T_i \leq T$ is satisfied. If this constraint is satisfied, then the global constraint is not violated. The coordinator need not poll the rest of the sites and hence this method reduces the communication overhead. However, if the constraint is still violated, the coordinator polls another larger subset of sites and performs the same procedure. This procedure is continued until either the coordinator detects that the global constraint is not violated or all sites have been polled. While this method reduces the communication required, it introduces undesired latency in detecting constraint violations which might be unacceptable for many applications. Hence, we prefer to restrict the number of rounds of polling to 1 or 2.

### 6.3.3 Cost model

We now present the model that we use to estimate the communication cost of a distributed constraints violation detection scheme. The coordinator keeps an estimate $Y_i$ of each $x_i$ such that $Y_i \geq x_i$ at all times. Local alarms from a remote site $s_i$ are used to update $Y_i$. Once a remote site exceeds its local threshold, it sends any change in its local value to the coordinator. It is easy to extend our schemes to an approximate scheme where a remote site sends updates to the coordinator on "significant" changes in its value only (for example, if the value changes by $\Delta v$, where $v$ is the previous value sent to the coordinator). The coordinator estimate of the global sum is approximate by a factor of $\Delta$.

$$\begin{aligned} Y_i &= x_i \text{ for each } s_i \text{ that reports a local alarm} \\ &= T_i \text{ for each } s_i \text{ that has not reported anything} \end{aligned}$$

Whenever this estimate $\sum_i Y_i > T$, the coordinator initiates a global poll to know the $x_i$ values and check if the constraint is actually violated.

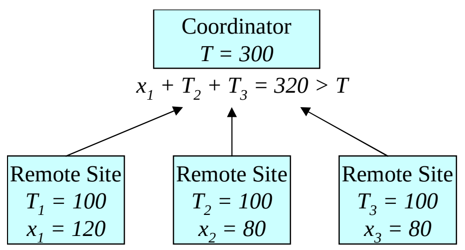
The communication cost of the system comes from local alarms and global polls. Define

- $P_l(i)$ : The probability of a local alarm at site $i = \Pr(x_i > T_i)$ - i.e. The probability that the value at remote site $s_i$ is greater than its threshold $T_i$.
- $P_g$ : The probability of a global poll $= \Pr(\sum_i Y_i > T)$

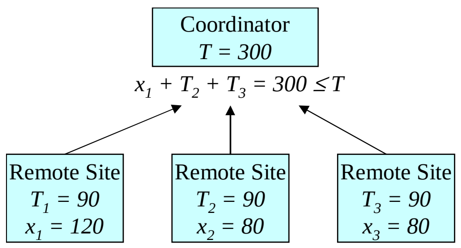
Let $C_l$ be the cost of transmitting a message from a remote site $s_i$ to the coordinator on a local alarm and $C_g$ be the cost of the global poll. Typically $C_l$ is $O(1)$ and $C_g$ is $O(n)$. Our model can be easily extended to systems where the costs of sending messages to the coordinator are different for the remote sites. For simplicity of exposition, we assume that this cost is the same for all remote sites. The communication cost of our scheme is then given by

$$C = P_g C_g + \sum_{i=1}^{n} P_l(i) C_l \tag{6.1}$$

Given a value of $T_i$ at a remote site, $P_l(i)$ depends entirely on the distribution of observed values at the remote site and is not affected by changes in behavior of the other sites (assuming that observed values are independent across sites). $P_g$, however, depends on the observed values as well as the threshold values at the other remote sites. Hence, a change in the threshold value at some site or a change in distribution of any site's observed value affects the $P_g$ across all sites.



**(a)** Assigning local thresholds of $T/n$ results in a global poll



**(b)** Assigning lower local thresholds absorbs transient spikes in values requiring no global polls

**Figure 6.1:** Example to show effect of slack in threshold assignment.

### 6.3.4 Local thresholds assignment problem

A simple method to set the local thresholds at the remote sites is to assign $T_i = T/n, \forall i \in [1, n]$. This is referred to as the *simple value* scheme in Reference [51]. Note that if some site $s_j$ violates its local constraint ($x_j > T/n$), then the coordinator site needs to perform a global poll. This is because $x_j + \sum_{i \neq j} T_i > T$. This method is an example of a zero slack scheme, i.e. $\sum_{i=1}^{n} T_i = T$. Note that in general for all zero slack schemes, a local alarm results in a global poll because the estimate of the constraint would exceed $T$ on a local alarm.

On the other hand, setting local thresholds to values less than $T/n$ provides some *slack* at the coordinator that can be used to avoid expensive global polls. The slack at the coordinator is given by $T - \sum_{i=1}^{n} T_i$ and any site can exceed its local threshold value by this value without needing a global poll. This slack can thus be used to absorb temporary spikes (e.g., flash crowds) in the values at the remote sites, whereas setting all local thresholds at $T/n$ would have resulted in global polls. Thus, a non-zero slack scheme can result in lower communication costs. Setting the local thresholds to very low values however results in frequent local alarms and hence high communication overhead. At an extreme, we can set all local thresholds to 0. This is equivalent to the remote sites sending every change in local values to the coordinator and hence, the coordinator tracks the exact values at the sites, requiring no global polls. However, the number of updates sent from remote sites to the coordinator is clearly unacceptably large.

We illustrate this with an example in Figure 6.1 that has 3 remote sites and has a threshold of $T = 300$. Figure 6.1a shows the case of a zero slack threshold assignment, where each local threshold is set to $T/n$ ($= 100$). The first site's value increases to 120, causing it to send a local alarm to the coordinator. The coordinator's site estimate ($x_1 + T_2 + T_3$) is greater than $T$ causing it to initiate a global poll. Figure 6.1b shows a non-zero slack assignment where the local thresholds are assigned lower than $T/n$ at 90. The slack in the system is 30 and since the first site's value is not more than this amount above its local threshold value, the coordinator does not need to initiate a global poll. This reduces the amount of communication required to track the distributed constraint. Note that if the local thresholds were set to 75 instead, this would have resulted in local alarms at all sites and the amount of communication would be equivalent to a global poll. Thus, it is vital to identify the optimum value of the local thresholds that results in low overall communication overhead. Reference [51] makes the same observation in their *Improved Value* scheme.

While smaller values of local thresholds results in higher probability of local alarms $P_l(i)$ and lower global poll probability $P_g$, larger values of local thresholds result in lower probability of local alarms and higher global polls probability. The optimum local threshold assignment balances these two costs to minimize the cost given by $C$ in Equation 6.1.

We now define the local threshold assignment problem.

**Problem Statement 7.** *Given a threshold $T$ and $n$ remote sites with values $x_i$ at site $i \in [1, n]$, determine the threshold values, $T_i$ at each site such the total cost of communication $C$, given by Equation (6.1), is minimized.*

### 6.3.5 Zipf case

We now present a more concrete example that illustrates the cost benefits of identifying optimal local thresholds. Consider $n$ remote sites, each having values in the range $[0, T]$ that follow a Zipf distribution (with Zipf exponent 1) in this example. The probability that site $s_i$ takes on a value $v$ is given by $Pr(x_i = v) = \frac{1}{H_T} . \frac{1}{v}$, where $H_i$ is the $i^{th}$ Harmonic Number defined by $H_i = \sum_{k=1}^{i} \frac{1}{k}$. Let the global threshold value be $T$, where $2^n < T < 2^{2^n}$. We assume that $C_l = 1$ for all sites and $C_g = n$.

**Theorem 8.** *For this example, if $C_{T/n}$ is the cost of the system when the local thresholds at all the sites are $T/n$ and $C_{T/\log(T)}$ is the cost of the system when the local thresholds at the sites are $T/\log(T)$, then the gain derived by using a threshold value of $T/\log(T)$ at the remote sites instead of $T/n$, given by $g = \frac{C_{T/n}}{C_{T/\log(T)}}$, is $\Omega(\log(n))$.*

*Proof.* The probability of local threshold violations is given by $P_l(i) = 1 - \frac{H_{T_i}}{H_T}$, the probability of the value at the site being less than $T_i$. Note that since $H_v$ is $\theta(\log(v))$,

$$P_l(i) = \frac{1}{H_T}\theta(\log(\frac{T}{T_i}))$$

For the case when the local thresholds are $T_i = T/n$, the global poll probability is the probability that none of the remote sites have a local threshold violation.

$$P_g(T_i = T/n) = 1 - (1 - P_l)^n \approx nP_l \text{ (assuming } P_l << 1)$$

Thus, the cost of the system when using local threshold values of $T/n$ at each remote site is given by

$$\begin{aligned} C_{T/n} &= nP_l(T_i = T/n)C_l + P_g(T_i = T/n)C_g = nP_l + n^2 P_l \\ &= \frac{n(n+1)}{H_T}\theta(\log(\frac{T}{T/n})) = \frac{n(n+1)}{H_T}\theta(\log(n)) \end{aligned}$$

When local thresholds are set to $T/\log(T)$, the global poll probability is that the probability that the estimate at the coordinator exceeds $T$, i.e. $Prob(Y = \sum_i Y_i > T)$. We use Markov's inequality to bound this probability and assume that each site's values are independently and identically distributed.

$$\begin{aligned} P_g(T_i = T/\log(T)) &= \Pr(Y = \sum_i Y_i > T) \\ &\leq \frac{E(\sum_i Y_i)}{T} = \frac{\sum_i E(Y_i)}{T} = \frac{nE(Y_i)}{T} \end{aligned}$$

Recall that $Y_i = T_i$ when $x_i < T_i$ and $Y_i = x_i$ otherwise. Thus,

$$\begin{aligned} E(Y_i) &= \sum_{j=0}^{T/\log(T)} T_i \Pr(x_i = j) + \sum_{j=T/\log(T)+1}^{T} x_i \Pr(x_i = j) \\ &= \frac{1}{H_T}\left(T + \frac{T}{\log(T)} H_{T/\log(T)} - \frac{T}{\log(T)} + 1\right) \end{aligned}$$

Note that $\frac{T}{\log(T)} - 1 > 0$ and

$$\frac{T}{\log(T)} H_{T/\log(T)} = \frac{T}{\log T}\theta\left(\log\left(\frac{T}{\log(T)}\right)\right) \leq T$$

Thus, $E(Y_i) \leq \frac{2T}{H_T}$ and the global poll probability is given by

$$P_g(T_i = T/\log(T)) \leq \frac{2n}{H_T}$$

The cost of the system when using local threshold values of $T/\log(T)$ at each remote site is given by

$$\begin{aligned} C_{T/\log(T)} &= nP_l(T_i = T/\log(T))C_l + P_g(T_i = T/\log(T))C_g \\ &= \frac{n}{H_T}\theta(\log(\frac{T}{T/\log(T)})) + \frac{2n^2}{H_T} \\ &= \frac{n}{H_T}\left(\theta(\log(\log(T))) + 2n\right) \end{aligned}$$

We now have the gain as

$$\begin{aligned} g = \frac{C_{T/n}}{C_{T/\log(T)}} &\geq \frac{(n+1)\theta(\log(n))}{\theta(\log(\log(T))) + 2n} \\ &= \Omega(\log(n)) \text{ since } T < 2^{2^n}, \log(\log(T)) < n \end{aligned}$$

□

We assume large values of $T$ ($> 2^n$) because violation of this threshold must be a rare event and hence the probability of this threshold being violated should be low. Note that since $T > 2^n$, the slack in the system is $T - \frac{nT}{\log(T)} > 0$. This slack absorbs the large values of the Zipf distribution at the sites and hence results in the cost gains.

This example shows that using a threshold value of $T/\log(T)$ results in a reduction in the communication overhead of tracking distributed constraint violations for the case where all sites' values follow a Zipf distribution. In reality, the values at the sites need not follow this distribution. Further, each site's values can follow a different distribution and the distribution of values can change over time. Thus, a threshold assignment algorithm should adapt to these changes when they happen.

## 6.4 Adaptive threshold assignment

We now present our algorithms for the problem of determining optimal local threshold values at the remote sites. We present three schemes to assign threshold values $T_i$ at the remote sites.

- **Brute force algorithm:** The first algorithm uses the coordinator to assist the remote sites to determine optimal threshold values. This algorithm performs well in our experiments. However, it requires each remote site to periodically send their histograms to the coordinator, and the coordinator performs a complex computation to determine the local thresholds for each remote site. This makes the algorithm relatively expensive and less desirable to use in large scale deployments.
- **Markov based algorithm:** This algorithm uses Markov's inequality to approximate the global poll probability and this results in a decentralized algorithm. The advantage of this algorithm is that it is decentralized except for a few messages to ensure correctness. It, however, performs relatively poorly in our experiments.
- **Reactive algorithm:** The third algorithm uses local alarm and global poll events in the system to assign the local thresholds at the remote sites. This algorithm does not require as much communication as the brute force algorithm but still results in comparable performance.

Before describing our algorithms, we present the cost of the geometric scheme [158] as applied to our problem. We consider this algorithm to be state of the art and compare the performance of our algorithms with this method. For all algorithms, a local alarm is a single message from the remote site to the coordinator and each global poll requires $n$ messages assuming that the coordinator polls every remote site.

### 6.4.1 Geometric approach

The geometric approach is an adaptive algorithm that we have described in Section 6.2. The communication costs of this scheme are as follows.

**Communication costs:** The cost for this scheme comprises of global polls and *control messages*. Following every global poll, the scheme sends $n$ *control messages*, one for each remote site to set new threshold values. We can ignore the cost due to local alarms since this is a zero-slack scheme and therefore the coordinator needs to perform a global poll for every local alarm.

### 6.4.2 Brute force algorithm

This algorithm uses information from all remote sites at the coordinator to compute the local threshold values. It determines the local thresholds by computing $P_l(i)$ and $P_g$, and then selecting the local threshold that minimizes the total cost $C$ (given by Equation 6.1).

Each site maintains a histogram of the values that it sees over time as $H_i(v), \forall v \in [0, T]$, where $H_i(v)$ is the probability of site $s_i$ taking the value $v$.

The probability of a local alarm is entirely dependent on the state of the remote sites and each remote site can independently calculate $P_l(i)$ at a given value of $T_i$ and is given by

$$P_l(i) = 1 - \sum_{j=0}^{T_i} H_i(j)$$

$P_g$ however is dependent on the state of all remote sites. In order to compute $P_g$, each remote site sends its local histogram to the coordinator periodically. We call this period the *recompute interval*. The coordinator uses the histograms to compute $P_g$.

$$P_g = \Pr(Y > T) = 1 - \sum_{v=0}^{T} \Pr(Y = v)$$

$\Pr(Y = v)$ can be computed at the coordinator using the following dynamic programming algorithm. Let $Q_i(v)$ denote the probability of the estimate of a remote site $s_i$'s value being equal to $v$; i.e. $\Pr(Y_i = v)$. Let $\zeta(k, v)$ denote $\Pr\left(\sum_{i=k}^{n} Y_i = v\right)$. Assuming that values at the sites are

```
BruteForce($T, n, \delta$)
1: $T_i \leftarrow \frac{T}{n}, \forall i \in [1, n]$;
2: loop {receive histograms from all remote sites every recompute interval;}
3:   Slack, $S \leftarrow T - \sum_{i=1}^{n} T_i$;
4:   $\delta_{upper} = min(\delta, \frac{S}{n})$;
5:   for $i = 1$ to $n$ do
6:     $U_i \leftarrow T_i + \delta_{upper}$; $L_i \leftarrow max(T_i - \delta, 0)$;
7:     $T_i(opt) \leftarrow T_i$; $C_i(opt) \leftarrow \infty$;
8:     for $\hat{T}_i = L_i$ to $U_i$ do
9:       Compute cost, $C_i$, at $\hat{T}_i$;
10:      if $C_i < C_i(opt)$ then
11:        $T_i(opt) \leftarrow \hat{T}_i$; $C_i(opt) = C_i$;
12:      end if
13:    end for
14:  end for
15:  $\forall i \in [1, n], T_i = T_i(opt)$;
16:  Send $T_i$ values to the remote sites.
17: end loop
```

**Figure 6.2:** Brute Force Algorithm

independent across sites (i.e. the $Q_i$'s are independent), we have:

$$Q_i(v) = \begin{cases} 0 & \text{if } v < T_i, \\ \sum_{v \leq T_i} H_i(v) & \text{if } v = T_i, \\ H_i(v) & \text{Otherwise.} \end{cases}$$

$$\zeta(k, v) = \begin{cases} Q_n(v) & \text{if } k = n, \\ \sum_{y \leq v} \zeta(k+1, v-y) * Q_k(y) & \text{if } k < n. \end{cases}$$

$\zeta(1, v)$ gives us $\Pr(Y = v)$, and $P_g$ can be computed by running the algorithm for each value of $v \in [0, T]$. Note that the same dynamic programming table can be used to compute $\Pr(Y = v)$ for all values of $v \in [0, T]$. The running time to compute the table is $O(nT^2)$ and to sum the entries ($\sum_{v=0}^{T} \zeta(1, v)$) takes $O(T)$ time and thus the algorithm has pseudo-polynomial complexity.

In order to determine the optimal threshold values at each site that result in minimum cost, we can do a naive exhaustive enumeration of all $T^n$ possible sets of local threshold values. For each combination of threshold values, we compute the $P_l(i)$ values at each site and the $P_g$ value to determine the cost. Thus, this naive enumeration has a running time of $O(nT^{n+2})$. This is clearly not scalable for large values of $T$ and $n$. We propose the following optimizations to make the running time of the computation manageable.

- While determining optimal threshold values, we calculate the optimal threshold value for a remote site by fixing the threshold value at the other sites. In other words, each site assumes that the other sites do not change their threshold values for the next round. Thus, we perform a search for the optimal threshold independently for each site and the complexity of this search is $O(nT)$ (compared to $O(T^n)$ search complexity for the exhaustive enumeration).

- Note that we assume that the other sites do not change their threshold values while calculating the optimal threshold value at a remote site. This is not true and the other sites can change their threshold values arbitrarily in the range $[0, T]$. If we allow arbitrary changes in the threshold values, the estimated $P_g$ value can be arbitrarily off from the actual $P_g$ value in the system . In order to prevent this, we assume that each remote site $i$ can vary its threshold value in the range $[T_i - \delta, T_i + \delta]$ only. Thus, we limit the error in the $P_g$ estimate at each site. This also reduces the search space for threshold values at each remote site to $2\delta$ from $T$.

- For the small range $[T_i-\delta, T_i+\delta]$, we also assume that $P_g$ is linear between the two endpoints. Thus, it sufficient to run the dynamic programming algorithm at the two end points only and at all other points in that range, we can calculate $P_g$ using the linear interpolation.

We present the algorithm with the optimizations in Figure 6.2. Note that $C_i$ represents the cost of the system assuming that only site $i$s threshold value changes in a given round.

**Ensuring correctness:** The slack computed in Line 3 of the algorithm is used to ensure that each remote site's threshold does not increase by a value more than $\delta_{upper}$ computed in Line 4. This ensures that $\sum_{i=1}^{n} T_i \leq T$, otherwise we will not be able to detect constraint violations correctly.

**Communication costs:** Apart from local alarms and global polls, each remote site sends an update (see Section 6.4.5 for details) of its histogram values every recompute interval and the coordinator recomputes the threshold values. Thus, there are $2n$ control messages in the system every recompute interval.

Note that we count the message to send histogram data from a remote site to the coordinator as one control message. This message is however larger in terms of size than a control message used to send new threshold values to remote sites. Thus, our estimate in terms of the number of messages (and not the size of messages) is an optimistic estimate of the control overhead of this algorithm.

### 6.4.3 Markov-based algorithm

The brute force algorithm requires remote sites to send their histograms every recompute interval and requires the coordinator to perform the above computation to determine the local threshold values. This is not very desirable when the number of remote sites is large. Our Markov-based algorithm decentralizes the computation of $P_g$ thus enabling each site to independently determine the local threshold values.

Using Markov's inequality,

$$P_g = \Pr(Y > T) \leq \frac{E[Y]}{T} = \frac{E[\sum_{i=1}^{n} Y_i]}{T} = \frac{\sum_{i=1}^{n} E[Y_i]}{T}$$

Thus, the cost of the system is given by

$$C = \sum_{i=1}^{n} C_l P_l(i) + C_g P_g \leq \sum_{i=1}^{n} C_l P_l(i) + \frac{C_g}{T} \sum_{i=1}^{n} E[Y_i]$$
$$C \leq \sum_{i=1}^{n} \left( C_l P_l(i) + \frac{C_g}{T} E[Y_i] \right)$$

$E[Y_i]$ can be computed at the local sites as follows.

$$E[Y_i] = \sum_{v=0}^{T} Y_i \Pr(Y_i = v) = \sum_{v=0}^{T_i} T_i H_i(v) + \sum_{v=T_i+1}^{T} v H_i(v)$$

Note that each site can independently determine the local threshold value that minimizes its contribution to the total cost, $C_l P_l(i) + \frac{C_g}{T} E[Y_i]$, thus requiring no assistance from the coordinator. The global poll probability $P_g$ using the Markov inequality is an upper bound on the actual probability and this estimate grows to 1 very quickly with increasing $T_i$ values. Hence, this algorithm assigns local threshold values that are much smaller than the optimum threshold values to minimize this estimated cost. This results in a large number of local alarms and hence, higher cost (other estimates using Hoeffding and Chebyshev bounds do not yield better results). We demonstrate this in our experiments in Section 6.5.

Each remote site computes its optimal threshold value every recompute interval. The optimal local thresholds are computed by performing a linear search in the range 0 to $T$. This takes $O(T)$ running time. We can reduce the running time to $O(\delta)$ by searching for the optimal threshold value in a small range $[T_i - \delta, T_i + \delta]$ in each round.

**Ensuring correctness:** If each remote site is allowed to independently decide on their local threshold values, we will not be able to ensure correctness; i.e. $\sum_{i=1}^{n} T_i \leq T$ cannot be guaranteed. A simple method to ensure correctness is to restrict each remote site's local to a maximum of $T/n$. This however can result in poor performance in cases where one site's value is very high on average compared to other sites.

In order to ensure that the sum of the threshold values is bounded by $T$, each remote site sends the computed optimal local threshold value, $T_i$, to the coordinator every recompute interval. The coordinator determines if $\sum_{i=1}^{n} T_i \leq T$. If not, it reduces each threshold value $T_j$ by $\frac{T_j}{\sum_{i=1}^{n} T_i} (\sum_{i=1}^{n} T_i - T)$. This ensures that $\sum_{i=1}^{n} T_i \leq T$.

**Communication costs:** Apart from local alarms and global polls, the Markov based algorithm sends $2n$ control messages every recompute interval to ensure correctness. Each remote site sends its calculated threshold value and the coordinator sends either modified threshold values or validates the threshold values calculated by the remote sites. These control messages are very light weight as compared to the control messages sent by the remote sites in the brute force algorithm.

```
LocalAlarmAction($i, T, n, \alpha, \rho_i$)
1: With probability $\min(1, \frac{1}{\rho_i})$, $T_i \leftarrow \alpha \times T_i$
```

```
GlobalPollAction($i, T, n, \alpha, \rho_i$)
1: With probability $\min(1, \rho_i)$, $T_i \leftarrow \frac{T_i}{\alpha}$
```

**Figure 6.3:** Reactive threshold assignment algorithm

### 6.4.4 Reactive algorithm

The reactive algorithm adjusts local threshold values at the remote sites based on local alarm and global poll events that occur in the system. Each local alarm from a remote site indicates that the threshold value at the remote site is possibly lower than optimum and each global poll indicates that the threshold value is higher than optimum. The basic reactive scheme at each remote sites adapts the thresholds at the remote sites based on these events using the algorithm shown in Figure 6.3.

Whenever there is a local alarm, the site increases the threshold value by a factor $\alpha$ with a probability $1/\rho_i$ (or 1, if $1/\rho_i$ is greater than 1), where $\alpha$ and $\rho_i$ are parameters of the system greater than 0. Whenever there is a global poll each remote site reduces the threshold value by a factor of $\alpha$ with a probability $\rho_i$ (or 1, if $\rho_i$ is greater than 1). $\alpha$ is a constant that determines the rate of convergence and can typically take values in the range $(1, 1.2]$. Choosing an $\alpha$ value that is too small leads to bad performance since it does not converge fast enough, while choosing a large $\alpha$ leads to large oscillations in threshold values. We choose $\alpha = 1.1$ in our experiments (which we found to be the best).

If we want the remote site to take the optimal local threshold $T_i^{opt}$, then setting $\rho_i = \frac{P_l(T_i^{opt})}{P_g^{opt}}$ will achieve this - here $P_l(T_i^{opt})$ is the probability of a local alarm when the local threshold is $T_i^{opt}$ and $P_g^{opt}$ is the probability of a global poll when all remote sites take the optimal threshold values. The reason for this is that if the system is not at $T_i^{opt}$ at all sites, then we can show that at some site either (1) current threshold $T_i' > T_i^{opt}$, $P_l(T_i') < P_l(T_i^{opt})$ and $P_g(T_i') > P_g(T_i^{opt})$, or (2) current threshold $T_i' < T_i^{opt}$, $P_l(T_i') > P_l(T_i^{opt})$ and $P_g(T_i') < P_g(T_i^{opt})$. Let us look at the first case; if at some site $T'_i > T_i^{opt}$, $P_l(T'_i) < P_l(T_i^{opt})$ and $P_g(T'_i) > P_g(T_i^{opt})$:

$$\frac{P_l(T'_i)}{P_g(T'_i)} < \frac{P_l(T_i^{opt})}{P_g(T_i^{opt})} \text{ and } P_l(T'_i) < \rho_i P_g(T'_i)$$

Hence, the average number of observed local alarms is lesser than $\rho_i$ times the average number of observed global polls. Thus, the threshold value decreases over time from $T'_i$. We can similarly argue that the threshold value will increase if the threshold is lesser than $T_i^{opt}$. So, the stable state of the system is around $T_i^{opt}$ using the reactive algorithm. This argument ignores other interactions in the system such as other sites varying their thresholds, thus affecting the observed $P_g$. We conjecture that the system converges to the desired value of $T_i$ even in the presence of these interactions and our experiments corroborate this. Observe that, we introduce randomness by increasing and reducing thresholds probabilistically and this desynchronizes threshold changes at the remote sites and helps in convergence.

Since determining the optimum $T_i^{opt}$ and the corresponding $P_g$ values is not feasible, we propose to use the Markov-based scheme to identify the threshold value that gives the minimum cost estimate and use this value to compute the contribution of the remote site to $P_g$. Every recompute interval, the remote site then sends this component of $P_g$ to the coordinator. The coordinator sums the components of $P_g$ it receives from the remote sites and computes the $P_g$ value. The coordinator sends this value of $P_g$ to the remote sites. Each remote site uses this value of $P_g$ to compute the value of $\rho_i$ for the reactive scheme. Note that the $P_l$ used in the computation of $\rho_i$ is for the threshold value that gives the minimum cost according to the Markov-based scheme.

**Comparison with Markov algorithm:** The Markov algorithm does not perform well because it sets the local thresholds to very low values. However, in the reactive scheme, the remote sites see far less global polls than is estimated by the Markov scheme and hence, sets higher threshold values than the Markov scheme. Thus, the reactive scheme is able to perform much better than the Markov scheme in our experiments.

let $T_i^{est}$ be the threshold value at remote site $i$ determined by the Markov scheme and $T_i^{real}$ be the threshold value where the system actually converges. Let $est(P_g)$ be the Markov estimate of the global poll probability, $P_g(T_i^{real})$ be the real global poll probability observed in the system at $T_i^{real}$ and $P_g(T_i^{est})$ be the real global poll probability at $T_i^{est}$. Also, we have that $P_g(T_i^{est}) \leq est(P_g)$ since Markov overestimates $P_g$. By definition,

$$\rho_i = \frac{P_l(T_i^{est})}{est(P_g)} = \frac{P_l(T_i^{real})}{P_g(T_i^{real})}$$

If $T_i^{real} < T_i^{est}$, we have:

$$P_l(T_i^{real}) > P_l(T_i^{est}) \text{ and } P_g(T_i^{real}) < P_g(T_i^{est}) \leq est(P_g)$$

Thus,

$$\frac{P_l(T_i^{est})}{est(P_g)} < \frac{P_l(T_i^{real})}{P_g(T_i^{real})}$$

Hence, we have a contradiction. Thus, $T_i^{real} \geq T_i^{est}$. In reality, Markov's estimate of $P_g$ is much higher than the real $P_g$ observed in the system and hence, the system converges to a threshold $T_i^{real}$ that is significantly higher than the threshold $T_i^{est}$ determined by the Markov-based algorithm.

**Ensuring correctness:** The coordinator is always aware of the latest threshold value at each remote site - this is because every time there is a local alarm, the remote site informs the coordinator of the value that caused the local alarm along with the new threshold value at that remote site. Therefore, whenever local alarms cause $\sum_{i=1}^{n} T_i > T$ at the coordinator, the resulting global polls reduce thresholds until $\sum_{i=1}^{n} T_i \leq T$.

**Communication costs:** Apart from local alarms and global polls, every recompute interval the scheme sends $2n$ control messages. Each site communicates its contribution to the global poll probability at its estimate of the optimum based on the Markov-estimate, the coordinator then adds up all these estimates from the remote sites and broadcasts this value to all the remote sites which then use it to compute the local $\rho$ value.

While our reactive scheme is inspired by the scheme presented by Olston *et al.* [136], our problem is very different from theirs. We address the differences in Section 6.2

### 6.4.5 Maintaining histograms

Our algorithms rely on histograms of the values at remote sites to determine optimal local thresholds. We can assume all values greater than $T$ at a remote site to be equal to $T$ without affecting the constraint being monitored. However, if we assume that the range of values at remote sites is $[0, T]$, we need a histogram size of $T$ and this clearly does not scale with increasing values of $T$.

We use equi-depth histograms at each monitoring site to keep track of the data distribution. We did experiments with varying histogram sizes and as results in subsequent sections will show, histograms that represent only 5% of the domain from which site values are drawn are sufficient to see the claimed significant savings in communication cost. In all experiments, we use exponential aging to ensure that the histogram reflects recently seen values more prominently than older ones. In practice, one could use more sophisticated histogramming techniques such as the ones in [65].

In order to reduce the control overhead associated with the brute force algorithm, we send site histograms to the coordinator only if the KL-distance [75] between the last shipped histogram and the current histogram exceeds a certain threshold. See [75] for efficient algorithms to compare distributions and estimate information theoretic distances.

### 6.4.6 Computational Overhead

The computation overhead has two components - the cost incurred at each remote site and the cost incurred at the central coordinator. If we maintain full histograms at each remote site, then the computational cost for each scheme is as follows (for each recompute interval):

- Markov based scheme: $O(T)$ cost at each remote site, $O(n)$ cost at the coordinator
- Brute force scheme: $O(T)$ cost at each remote site, $O(nT^{n+2})$ at the coordinator. With the approximations we suggest the computational overhead at the coordinator reduces to

$O(n^2T^2 + n\delta)$ (where threshold values are allowed to vary in a O($\delta$) range). Since we expect O($\delta$) to be small, the cost at the coordinator for this scheme is $O(n^2T^2)$.

- Reactive scheme: $O(T)$ cost at each remote site, $O(n)$ cost at the coordinator.

As expected, the brute force scheme has a huge computational overhead and is intractable for asymptotic $n$ and $T$. The other schemes are less computationally intensive. Note that although the cost depends on $T$ since the comparison assumes we use full-size histograms; in practice however, we will only use approximate histograms. As mentioned in Section 6.4.5, we were able get very good results in practice even with tiny histograms.

## 6.5 Experiments

We performed extensive experiments, using multiple real-world traces and also using synthetic data, to evaluate the performance of our non-zero slack schemes and to explore properties of our algorithms. When using our schemes to set thresholds at monitors for both real-world data and synthetic data, we observed significant savings (40% to 90%) in the number of messages, over using the state of the art zero-slack geometric scheme in [158]. We also found that the savings in communication overhead when using our non-zero slack schemes increases as the number of monitoring nodes in the system increases. Our experimental results indicate that among the non-zero slack schemes that we suggest, the reactive scheme is the best in terms of performance across different datasets and also in terms of scalability.

### 6.5.1 Experimental Setup

For our experiments, we consider a monitoring application that monitors the total amount of traffic flowing into a service provider network. Our monitoring setup consists of getting information about the ingress traffic of the network. This information can be derived by deploying passive monitors ([59], [44]) at each link or by collecting flow information (e.g. Netflow [34] records) from the ingress routers. Each monitor aggregates the information (packet level or flow level) to derive the total amount of traffic (in bytes in this experiment) coming into the network through that ingress point. The distributed constraint monitors the total amount of traffic flowing into the network across all ingress links and throws an alarm if this amount exceeds a certain threshold.

In our experiments, we use the number of messages sent by each scheme to track the constraint as the performance metric. We used a recompute interval of 1000 rounds for each scheme. For the bruteforce scheme, we used $\delta = 10$. For the reactive scheme, we used $\alpha = 1.1$. For all schemes we used histograms that represent 5% of their original range. Refer to Section 6.4.5 for details on histogram computation and maintenance.

### 6.5.2 Datasets

**Abilene:** Our first set of Netflow traces were obtained from the Abilene network, which is an Internet2 high-performance backbone network with 11 routers located across the US [3]. We used traces from +1000hrs to +1100hrs UTC on August 15th, 2006. The Netflow records show a total of 73.3 million packets during this period. These packets were seen across the 11 Abilene router nodes; the Chicago location saw the most packets (9.6 million) while the Seattle location saw the least packets (2.2 million). Therefore the Abilene dataset gave us a real-life, large scale and *naturally* distributed dataset. We scaled down all packet sizes by a factor of 100. Although the results we present are for this selected time frame, we performed several experiments by looking at data that spanned weekend/weekday transitions, different days of the week as well as different times of day. We obtained very similar results for all these cases.

**NLANR:** We also use publicly available link traces from NLANR [133] as input to our distributed monitoring system. This trace was collected with an NLANR PMA OC192MON located on SDSC's TeraGrid Cluster, from +0000hrs UTC to +0100hrs UTC on February 18th, 2004. The trace contained a total of 21 million packets. These traces are for a single ingress link, and we transform this data for our distributed system by assigning a probability distribution for distributing packets randomly to the various monitors. By using different probability distributions, we can simulate various scenarios that can occur in real networks. A uniform distribution implies that any packet is equally likely to go to any of the monitoring nodes. A skewed distribution distributes packets unevenly and a few nodes receive more packets than others. For ease of presentation and also because we have a dataset that is naturally separated in Abilene, we only present the case where data is uniformly distributed across the different routers in this case. We also present results for experiments where we vary the number of monitoring nodes (from 10 to 160) across which this data is distributed. We scaled down all packet sizes by a factor of 10.

**Synthetic:** Finally, we also used synthetic datasets to evaluate parameters that could not be controlled using real-world datasets. For instance, to study the effect of changing the number of monitors (from 10 to 160) on the observed savings. Values at each monitor were generated independently at random in the range $[0, 1500]$ using a Zipf distribution with a randomly chosen Zipf exponent between 1.0 and 2.0. We ran experiments where all the monitors had the same behavior as well as when they all had different behavior. Unless specifically mentioned, we use $n = 20$ monitors whenever we mention that we generate synthetic data.

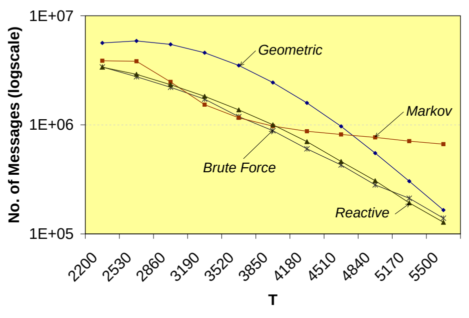
### 6.5.3 Results

**Comparison of message overhead:** Figure 6.4a compares the number of messages sent by the various threshold setting mechanisms, while Figure 6.4b compares the percentage gain in number of messages of the various schemes over using the geometric scheme for the Abilene dataset. In each plot we vary the threshold values ($T$) on the x-axis in a way that we have the true global poll percentage (i.e. the percentage of the number of true global constraint violations) vary from 50% to 0%. Notice that we get a 40%-90% improvement over the geometric scheme using our reactive scheme. Also notice that the reactive scheme performs close to the bruteforce approach throughout the range of $T$ values. Notice that as the event we are tracking becomes rare, the performance of the Markov scheme degrades rapidly and it actually performs much worse than the geometric scheme. Our reactive scheme does not suffer from these drawbacks. This is because Markov overestimates global poll probability and sets lower thresholds, while (as argued earlier) in the reactive case, thresholds converge to a higher value. These savings in plots *include* the various costs incurred by the schemes in computing the thresholds and communicating them to the monitors. One explanation for the drop off in gain while tracking rare events is that the control overhead of schemes begin to dominate the overall cost since the number of local alarms and global polls in these cases is small. Figures 6.4c and 6.4d show similar plots for the NLANR dataset. Figures 6.4e and 6.4f show similar results for the synthetic dataset.

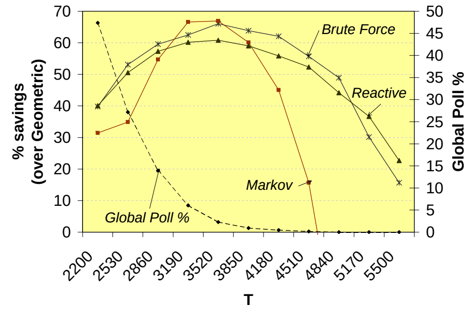
As we mentioned earlier, while the bruteforce approach shows very good performance it involves periodically shipping site-histograms and therefore may not be very practical. We use its performance more as an indicator of the optimum performance (that a histogram-based scheme might achieve).

**Breakdown of message overhead:** Figure 6.5 shows the breakup of cost for the various schemes while tracking events in the Abilene dataset. Each stacked chart breaks down the message overhead in terms of local alarms, global polls and control overhead. In general, if a scheme sets lower thresholds than another, then it will cause more local alarms than global polls compared to the other scheme. The Markov scheme (Figure 6.5c) (especially at higher values of T - i.e. when tracking rare events), overestimates the global poll probability and sets very low thresholds. As a result, the number of global polls is very low. However, the scheme suffers due to the large number of local alarms. The reactive scheme (Figure 6.5d) and the bruteforce scheme (Figure 6.5a) are able to strike a good balance here and this is reflected in their performance - where these two schemes consistently perform the best in the entire range of $T$ values (Figure 6.4a). The cost of the geometric scheme (Figure 6.5b) is comprised entirely of global polls and control overhead - this is because it maintains zero slack and consequently every local alarm results in a global poll. The bruteforce scheme has relatively low control overhead because we employ KL-distance based histogram shipping. Another interesting observation is that at higher values $T$, control overhead begins to look significant due to the smaller number of local alarms and global polls at these values.

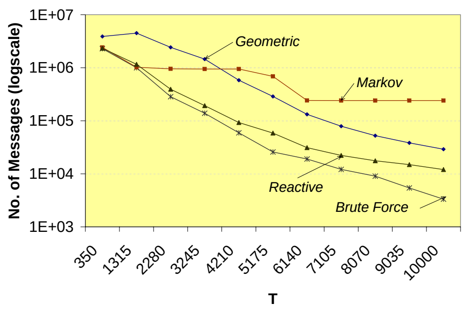
**Sensitivity to histogram size:** As mentioned in Section 6.4.5, we did all our experiments using small equi-depth histograms. We did experiments to see just how small we could make our histograms without significantly increasing the number of messages sent by our schemes by varying



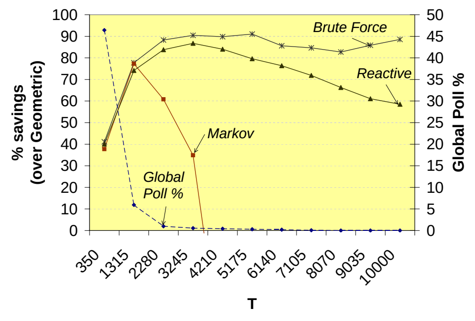
**(a)** Abilene dataset - number of messages



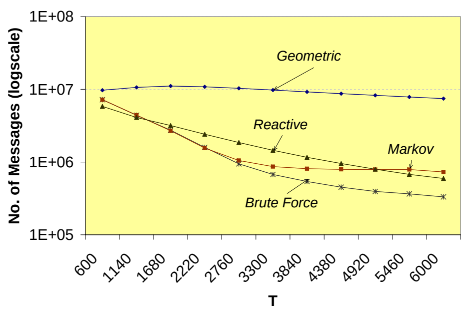
**(b)** Abilene dataset - percentage savings



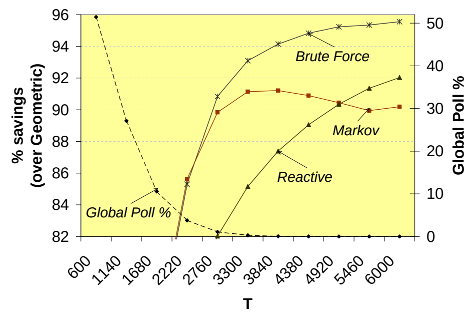
**(c)** NLANR dataset - number of messages



**(d)** NLANR dataset - percentage savings



**(e)** Synthetic data - number of messages

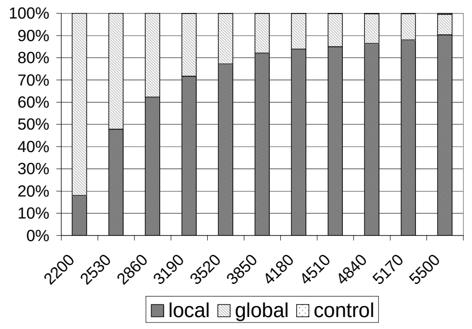


**(f)** Synthetic data - percentage savings

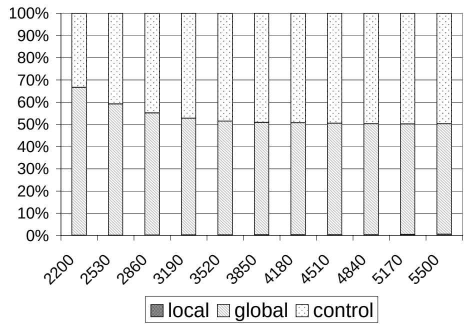
**Figure 6.4:** Number of messages sent in tracking events at various value of $T$ and Percentage Savings in number of messages over the geometric scheme.

their size from 20% of the original range to 1% of the original range. Figure 6.6, shows the results for the Abilene and synthetic datasets while tracking three different events that occur 5% of the time, 1% of the time and an event that occurs 0.0005% of the time. As can be seen in Figure 6.6, there is no significant change in the number of messages that each scheme sends until histograms shrink to around 1% of their original size. We obtained similar results for the NLANR dataset.

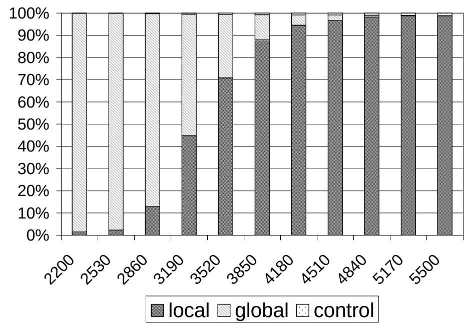
**Effect of scale - varying the number of monitoring nodes:** Our analytical result from



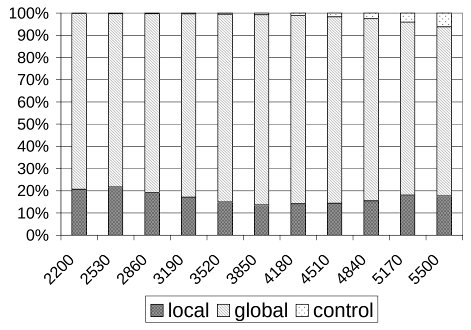
**(a)** Brute Force



**(b)** Geometric

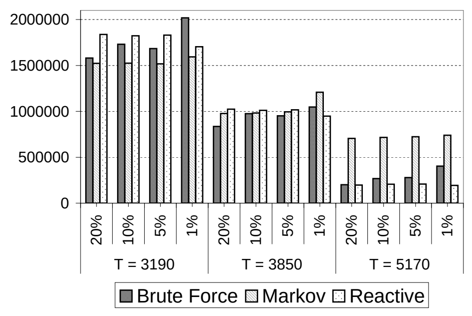


**(c)** Markov

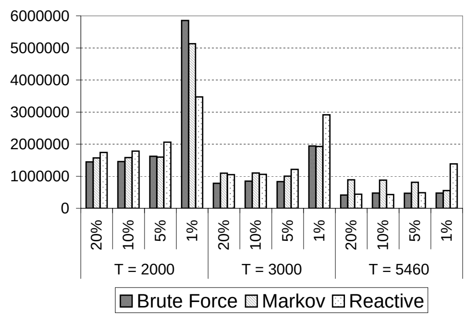


**(d)** Reactive

**Figure 6.5:** Bar charts showing the breakup of cost for the various schemes while tracking various events for the Abilene dataset.



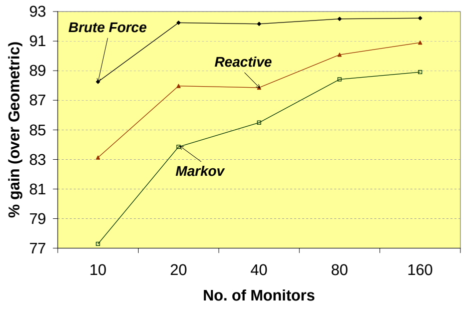
**(a)** Abilene



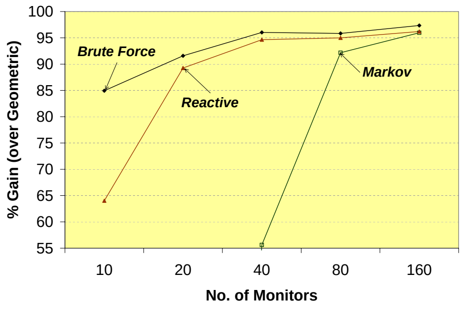
**(b)** Synthetic

**Figure 6.6:** Bar charts showing the variation in cost for the various schemes while using histograms of different sizes. Notice that the solution quality remains largely unaffected until histograms shrink to around 1% of their original size.

Theorem 8 suggests that the benefit from using a non-zero slack scheme over using a zero-slack scheme should increase as the number of monitoring nodes in the system increases. The theorem of course makes assumptions that might not always hold in practice. We did experiments to determine the savings in communication using non-zero slack schemes with increasing number of monitoring nodes. Our first experiment was with the synthetic dataset, where we varied the number of monitoring nodes from 10 to 160, and each site chose a random Zipf exponent in the range of 1.0 to 2.0. Our second experiment was with the monolithic NLANR dataset, where we varied the number of monitoring nodes from 10 to 160 and we distributed the packets in the trace by randomly assigning them to the monitors. In both experiments, we were interested in tracking an event that happens at most once during the entire trace. Figure 6.7a shows the results for the synthetic experiment and Figure 6.7b shows the results for the NLANR experiment. Notice that in both cases there is an increase in percentage gain over the geometric scheme as the number of monitoring nodes increases. The most interesting result from these experiments was that our Markov-based scheme starts performing very well for the NLANR dataset as the number of monitoring nodes increases. Notice that in 6.7b, the Markov-based scheme goes from performing much worse than the geometric scheme (this portion of the Markov plot is cut off) when $n = 10$, to showing a 95% savings in communication cost when $n = 160$. This is because in the NLANR dataset we keep the $T$ value constant even as we increase $n$ - since the event we are tracking remains the same in this case. Markov's estimate of the optimum threshold and the actual optimum threshold are actually close by because the range of good threshold values shrinks as $n$ increases and $T$ remains the same. We point out that while the bruteforce approach shows good results, its computation took a long time at high values of $n$.



**(a)** Zipf, T = 50n



**(b)** NLANR, T = 9035

**Figure 6.7:** Percentage gain over the geometric scheme as we vary the number of monitoring sites.

**Global poll probability comparison:** Predicting the correct global poll probability is the crucial part of determining the right thresholds to set to minimize communication cost. Figure 6.8b compares the global poll probability estimates using the bruteforce approach and using the Markov-based approach. The plots also include the observed global poll probability (calculated using the number of global polls seen in the system). The plots also indicate the true global poll probability (labeled "full-knowledge" in the plot). Notice that in Figure 6.8b, for the bruteforce scheme, the predicted global poll probability and the observed global poll probability are close to each other, this is a good indicator that the global poll probability estimates that the scheme uses are fairly accurate. Also notice that while the observed and estimated global poll probabilities are very different from each other for the Markov scheme, the observed global poll probability is actually lower than that of the bruteforce scheme. This is because the Markov scheme overestimates the global poll probability and sets a much lower threshold than is optimal. This ensures that Markov tracks the global sum accurately, but at the cost of increased communication in the form of local alarms - causing the Markov scheme to perform badly overall. The spike in the plot is just an artifact of the data - the sites see a lot of data in that period.

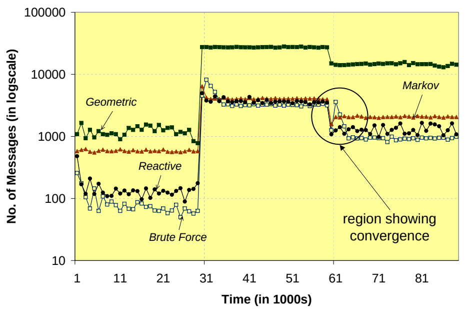
**Sensitivity to $\alpha$:** The $\alpha$ parameter in the reactive algorithm controls how quickly threshold values can be changed by the algorithm. Small values of $\alpha$ will ensure that thresholds cont change dramatically, while larger values of $\alpha$ will allow sites to change their thresholds quickly to adapt to changes in the distribution. Ideally, we should be able to pick a very small value of $\alpha$ and it should only affect the reactive scheme in as much as how fast it converges. However, in practice, choosing a very small $\alpha$ amounts to setting a minimum value on the threshold. For any threshold value less than $\frac{\alpha}{\alpha-1}$, we will require several global polls to see a significant reduction in the threshold value. It is therefore important to use an $\alpha$ value such that $\frac{\alpha}{\alpha-1}$ is small enough. This can be seen in Figure 6.8c, which shows the percentage difference in the number of messages between the best $\alpha$ value and a given $\alpha$ value for four different $T$ values for the Abilene dataset. Consider the case where $\alpha = 1.001$ and $T = 5500$ and the scheme performs badly, while at lower $T$ values and the same $\alpha = 1.001$, the scheme does not performs better. This is because at lower values of

$T$ there are enough global polls to cause a significant change in the threshold value. The reason for the drop off at higher values of $\alpha$ is simply that the thresholds change too rapidly even for passing anomalies or momentary spikes in the data. Our experiments showed similar results for the NLANR and synthetic datasets.

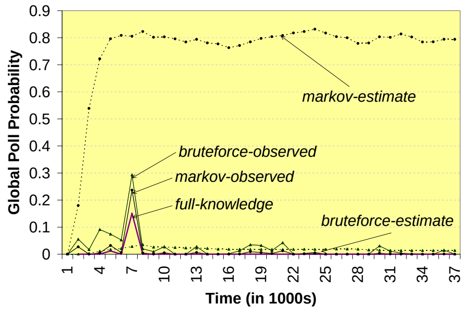
**Adaptability to changes in data distribution:** We performed an experiment to show that our schemes can adapt to changes in distribution even though they rely on aged histograms to calculate threshold values. We start with all monitors seeing data drawn from a Zipf distribution with a high Zipf exponent of 2.0. After 30000s, the distribution changes to a much lower Zipf exponent of 1.25. After another 30000s, the distribution changes to a higher Zipf exponent of 1.5. Figure 6.8a shows the number of messages sent by the various schemes in an interval spanning 1000s. Notice that in all cases, the bruteforce scheme performs the best, although it takes longer than the rest of the schemes to converge. The reactive scheme converges faster than the bruteforce approach and performs almost as well for both the high and low Zipf exponents. In Figure 6.8a, the circled region shows how the various schemes converge; while the Markov scheme and reactive scheme move steeply toward their best cost, the bruteforce takes longer to get to its best cost. This is because the bruteforce scheme adjust thresholds in additive increments. The reactive scheme on the other hand adjusts thresholds in multiplicative increments which leads to exponential convergence. The Markov scheme can make arbitrary changes to its threshold values.

**Sensitivity to $\rho$:** The value of $\rho$ determines the threshold values to which the reactive scheme will converge to. For instance if $\rho = 1$ then the reactive scheme will converge to a point where the local alarm probability is equal to the global poll probability. Clearly, for arbitrary distributions the expected cost need not be minimum at the point where they are equal. In general $\rho$ can be any value at the optimum point. For example, look at Figures 6.9a and 6.9b. The plots show the expected message overhead and the ratio of local alarms to global polls ($\rho$) as a function of slack for two different cases.

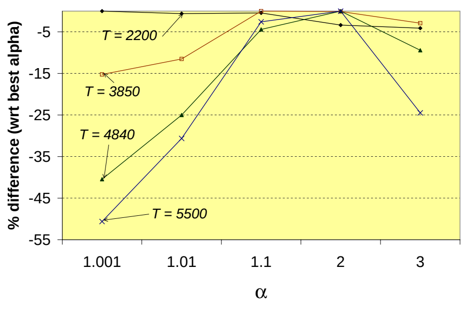
Notice that the optimum $\rho$ value is different in the two cases - the optimum is close to $\rho = 10$ in 6.9a, while its close to $\rho = 3$ in 6.9b.



**(a)** Plot showing how the different schemes adapt to a drastic change in distribution. The circled region shows that the various schemes converge at different rates when there is a change in distribution.



**(b)** Plot comparing the global probability estimates for the Markov-based scheme and the bruteforce scheme over a one hour trace from the Abilene dataset while tracking an event that occurs 0.5% of the time throughout the trace (this corresponds to $T = 4180$).



**(c)** Difference (in % messages) between a given $\alpha$ value (varied along the x-axis) and the best $\alpha$ value for 4 different values of $T$ for the Abilene dataset.

**Figure 6.8:** Additional experimental results

The reactive scheme we use in our experiments computes the $\rho$ value at the estimated optimum adaptively for each monitor using a Markov-based approach. We performed a set of experiments to determine how the reactive scheme performed if this $\rho$ value were to be changed. In these experiments, $\rho$ values ranging from 0.0001 to 10000 were chosen and fixed throughout the execution of the reactive scheme. The performance in each of these cases was noted for the entire

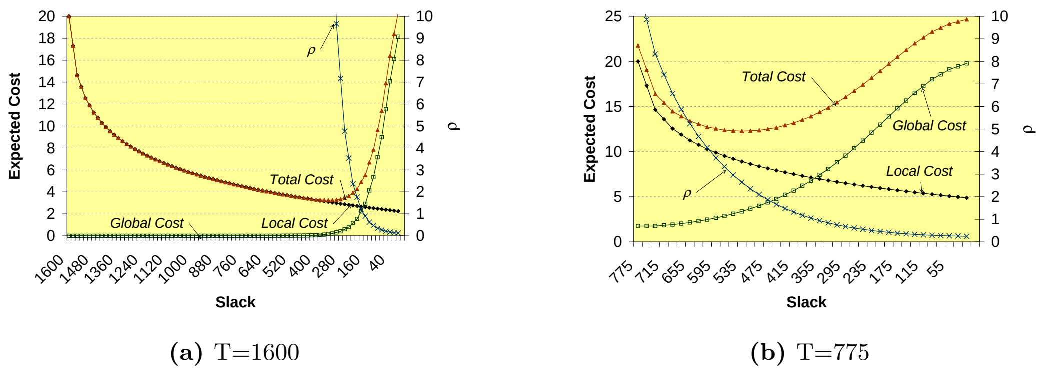
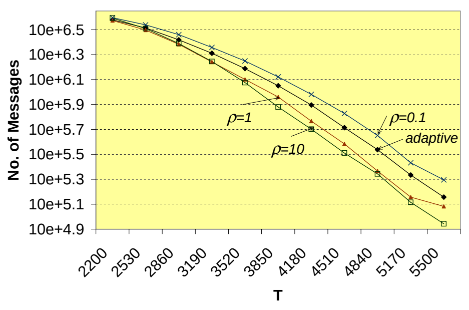


**Figure 6.9:** Expected cost curves and $\rho$ values (at two different T values) for an instance where n=20 and each monitor sees data drawn from a Zipf distribution with a Zipf exponent of 1.

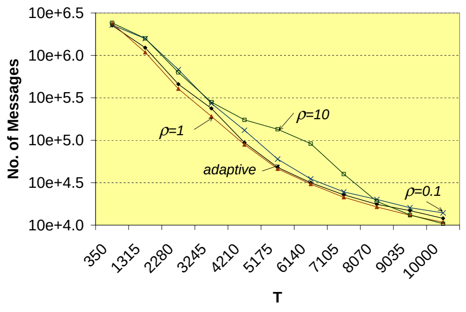
range of $T$ values that we examined earlier. The performance of these schemes over the range of $T$ values was compared to our scheme where the $\rho$ value was computed adaptively. See Figure 6.10 for results. The synthetic dataset for the results in Figure 6.10c were generated using $n = 20$ monitors where each site saw data drawn from a Zipf distribution with a Zipf exponent of 1. In all the experiments, the approach of computing $\rho$ adaptively performed close to the scheme with the best value of $\rho$ - note that of course, we do not know this optimum $\rho$ value in advance for a given dataset. Also as expected, the best $\rho$ value was different for different datasets. These experiments stress the importance of computing $\rho$ adaptively as opposed to fixing the $\rho$ value apriori. The main advantage however of fixing the $\rho$ value is that it completely avoids histogram maintenance and local computation. In our experiments, we found that choosing a $\rho$ value of 1 often yields reasonably good results - so if histogram maintenance and local computation is infeasible, then a variant of our reactive scheme that uses a fixed value of $\rho$ across all sites is an attractive alternative.

## 6.6 Conclusion and Future Work

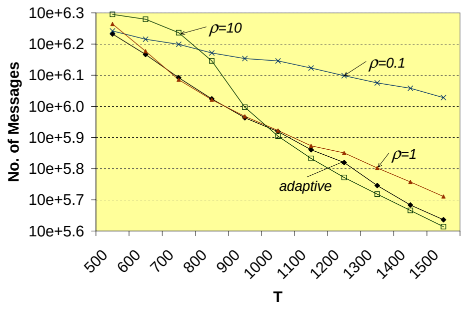
We have shown that non-zero slack algorithms result in better performance than state of the art zero slack algorithms in typical monitoring settings. We have presented three non-zero slack algorithms that adapt to changing distributions to efficiently monitor distributed constraints. The reactive scheme is lightweight in terms of communication overhead and still has solution



**(a)** Abilene



**(b)** NLANR



**(c)** Synthetic

**Figure 6.10:** Effect of using different values of $\rho$ (ratio of local alarms to global polls) for the reactive scheme. In all datasets, our approach of adaptively changing $\rho$ performs close to the scheme with the best (fixed) $\rho$ value. Also notice that the best performing $\rho$ value is different in the various datasets - intuitively, the $\rho$ value determines the "sweet spot" in the cost curve.

quality comparable to the heavyweight brute force algorithm. The reactive scheme is therefore an attractive method for practical use.

In this chapter, we have studied the implementation of a simple distributed constraint, sum of variables. It would be interesting to generalize the observation that non-zero slack methods can result in better performance for general functions (like join sizes, quantiles etc.) using the framework proposed by [158]. Reference [87] suggested a novel tracking problem called cumulative triggers and it would be interesting to see how our methods perform when applied to their problem. In typical networks, nodes can be organized in a hierarchical structure that can be exploited to further reduce communication required in implementing distributed constraints. Studying non-zero slack algorithms for such structured networks presents an interesting area of future research.

# Chapter 7

# Conclusions

This thesis studied three basic problems that arise in the context of self-management of both centralized and decentralized storage networks. Chapter 2 discussed the centralized data placement problem, Chapter 3 discussed the centralized data reconfiguration problem, Chapter 4 discussed the decentralized data placement and reconfiguration problems, Chapter 5 discussed the problem of answering one-shot queries and Chapter 6 discussed the problem of answering continuous queries. The following problems would constitute interesting extensions to the results and problems presented in this thesis:

1. A constant factor approximation algorithm for minimization version of the data placement problem with connection costs and bandwidth and storage constraints.

2. A constant factor approximation algorithm for the one-round migration problem defined on page 65.

3. Lower bounds for message and round complexity for aggregate computation using gossip-style communication.

4. Generalizing non-zero slack schemes for continuous monitoring of functions more general than sums and counts.

5. Generalizing the non-zero slack schemes presented in Chapter 6 to handle the case where the values observed at the sites being monitored are correlated across the sites.

# Bibliography

[1] http://www.pdos.lcs.mit.edu/chord/.

[2] D. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, C. Erwin, E. Galvez, M. Hatoun, A. Maskey, A. Rasin, A. Singer, M. Stonebraker, N. Tatbul, Y. Xing, R. Yan, and S. Zdonik. Aurora: a data stream management system. In *SIGMOD '03: Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 666–666, New York, NY, USA, 2003. ACM Press.

[3] Abilene Observatory. http://abilene.internet2.edu/.

[4] Micah Adler, Soumen Chakrabarti, Michael Mitzenmacher, and Lars Rasmussen. Parallel randomized load balancing. In *STOC '95: Proceedings of the twenty-seventh annual ACM symposium on Theory of computing*, pages 238–247, New York, NY, USA, 1995. ACM Press.

[5] Micah Adler, Eran Halperin, Richard M. Karp, and Vijay V. Vazirani. A stochastic process on the hypercube with applications to peer-to-peer networks. In *STOC '03: Proceedings of the thirty-fifth annual ACM symposium on Theory of computing*, pages 575–584, New York, NY, USA, 2003. ACM Press.

[6] Shipra Agrawal, Supratim Deb, KVM Naidu, and Rajeev Rastogi. Efficient detection of distributed constraint violations. *To appear in IEEE 23rd International Conference on Data Engineering (ICDE 2007)*, 2007.

[7] Yanif Ahmad, Bradley Berg, Uğur Cetintemel, Mark Humphrey, Jeong-Hyon Hwang, Anjali Jhingran, Anurag Maskey, Olga Papaemmanouil, Alexander Rasin, Nesime Tatbul, Wenjuan Xing, Ying Xing, and Stan Zdonik. Distributed operation in the Borealis stream processing engine. In *SIGMOD '05: Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pages 882–884, New York, NY, USA, 2005. ACM Press.

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

[9] Muneeb Ali and Koen Langendoen. A case for peer-to-peer network overlays in sensor networks. In *Int'l Workshop on Wireless Sensor Network (WWSNA'07), with 6th IPSN'07*, MIT Campus, Cambridge, MA, USA, April 2007.

[10] Noga Alon, Phillip B. Gibbons, Yossi Matias, and Mario Szegedy. Tracking join and self-join sizes in limited storage. In *PODS '99: Proceedings of the eighteenth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 10–20, New York, NY, USA, 1999. ACM Press.

[11] Noga Alon, Yossi Matias, and Mario Szegedy. The space complexity of approximating the frequency moments. In *STOC '96: Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*, pages 20–29, New York, NY, USA, 1996. ACM Press.

[12] Eric Anderson, Joseph Hall, Jason D. Hartline, Michael Hobbs, Anna R. Karlin, Jared Saia, Ram Swaminathan, and John Wilkes. An experimental study of data migration algorithms. In *WAE '01: Proceedings of the 5th International Workshop on Algorithm Engineering*, pages 145–158, London, UK, 2001. Springer-Verlag.

[13] Arvind Arasu, Brian Babcock, Shivnath Babu, Mayur Datar, Keith Ito, Itaru Nishizawa, Justin Rosenstein, and Jennifer Widom. STREAM: the stanford stream data manager (demonstration description). In *SIGMOD '03: Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 665–665, New York, NY, USA, 2003. ACM Press.

[14] James Aspnes and Gauri Shah. Skip graphs. In *SODA '03: Proceedings of the fourteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 384–393, Philadelphia, PA, USA, 2003. Society for Industrial and Applied Mathematics.

[15] Baruch Awerbuch and Christian Scheideler. Towards a scalable and robust DHT. In *SPAA '06: Proceedings of the eighteenth annual ACM symposium on Parallelism in algorithms and architectures*, pages 318–327, New York, NY, USA, 2006. ACM Press.

[16] Brian Babcock and Chris Olston. Distributed top-k monitoring. In *SIGMOD '03: Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 28–39, New York, NY, USA, 2003. ACM Press.

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

[18] M. Bawa, H. Garcia-Molina, A. Gionis, and R. Motwani. Estimating aggregates on a peer-to-peer network. In *Technical report, Computer Science Dept., Stanford University*, 2003.

[19] Michael W. Berry, Zlatko Drmac, and Elizabeth R. Jessup. Matrices, vector spaces, and information retrieval. *SIAM Review*, 41(2):335–362, 1999.

[20] Steven Berson, Shahram Ghandeharizadeh, Richard Muntz, and Xiangyu Ju. Staggered striping in multimedia information systems. In *SIGMOD '94: Proceedings of the 1994 ACM SIGMOD international conference on Management of data*, pages 79–90, New York, NY, USA, 1994. ACM Press.

[21] Bobby Bhattacharjee, Sudarshan Chawathe, Vijay Gopalakrishnan, Pete Keleher, and Bujor Silaghi. Efficient peer-to-peer searches using result-caching. In *The 2nd International Workshop on Peer-to-Peer Systems (IPTPS'03)*, February 2003.

[22] Indrajit Bhattacharya, Srinivas R. Kashyap, and Srinivasan Parthasarathy. Similarity searching in peer-to-peer databases. In *ICDCS '05: Proceedings of the 25th IEEE International Conference on Distributed Computing Systems (ICDCS'05)*, pages 329–338, Washington, DC, USA, 2005. IEEE Computer Society.

[23] Stephen Boyd, Arpita Ghosh, Balaji Prabhakar, and Devavrat Shah. Randomized gossip algorithms. *IEEE/ACM Trans. Netw.*, 14(SI):2508–2530, 2006.

[24] André Brinkmann, Kay Salzwedel, and Christian Scheideler. Efficient, distributed data placement strategies for storage area networks (extended abstract). In *SPAA '00: Proceedings of the twelfth annual ACM symposium on Parallel algorithms and architectures*, pages 119–128, New York, NY, USA, 2000. ACM Press.

[25] André Brinkmann, Kay Salzwedel, and Christian Scheideler. Compact, adaptive placement schemes for non-uniform requirements. In *SPAA '02: Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures*, pages 53–62, New York, NY, USA, 2002. ACM Press.

[26] Andrei Z. Broder, Moses Charikar, Alan M. Frieze, and Michael Mitzenmacher. Min-wise independent permutations (extended abstract). In *STOC '98: Proceedings of the thirtieth annual ACM symposium on Theory of computing*, pages 327–336, New York, NY, USA, 1998. ACM Press.

[27] Sirish Chandrasekaran, Owen Cooper, Amol Deshpande, Michael J. Franklin, Joseph M. Hellerstein, Wei Hong, Sailesh Krishnamurthy, Samuel R. Madden, Fred Reiss, and Mehul A. Shah. TelegraphCQ: continuous dataflow processing. In *SIGMOD '03: Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 668–668, New York, NY, USA, 2003. ACM Press.

[28] Moses S. Charikar. Similarity estimation techniques from rounding algorithms. In *Proceedings of the thiry-fourth annual ACM symposium on Theory of computing*, pages 380–388. ACM Press, 2002.

[29] Chandra Chekuri and Sanjeev Khanna. On multidimensional packing problems. *SIAM J. Comput.*, 33(4):837–851, 2004.

[30] Jen-Yeu Chen, Gopal Pandurangan, and Dongyan Xu. Robust computation of aggregates in wireless sensor networks: distributed randomized algorithms and analysis. In *IPSN '05: Proceedings of the 4th international symposium on Information processing in sensor networks*, page 46, Piscataway, NJ, USA, 2005. IEEE Press.

[31] Mitch Cherniack, Hari Balakrishnan, Magdalena Balazinska, Donald Carney, Ugur Cetintemel, Ying Xing, and Stan Zdonik. Scalable Distributed Stream Processing. In *CIDR 2003 - First Biennial Conference on Innovative Data Systems Research*, Asilomar, CA, January 2003.

[32] Ann Louise Chervenak. *Tertiary storage: an evaluation of new applications.* PhD thesis, University of California at Berkeley, Berkeley, CA, USA, 1994.

[33] C. F. Chou, L. Golubchik, and J. C.S. Lui. A performance study of dynamic replication techniques in continuous media servers. Technical Report CS-TR-3948, University of Maryland, October 1998.

[34] Cisco netflow. http://www.cisco.com/warp/public/732/Tech/netflow.

[35] Edith Cohen and Scott Shenker. Replication strategies in unstructured peer-to-peer networks. In *SIGCOMM '02: Proceedings of the 2002 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 177–190, New York, NY, USA, 2002. ACM Press.

[36] Jeffrey Considine, Feifei Li, George Kollios, and John Byers. Approximate aggregation techniques for sensor databases. In *ICDE '04: Proceedings of the 20th International Conference on Data Engineering*, page 449, Washington, DC, USA, 2004. IEEE Computer Society.

[37] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms.* The MIT Press, 1989.

[38] Graham Cormode and Minos Garofalakis. Sketching streams through the net: distributed approximate query tracking. In *VLDB '05: Proceedings of the 31st international conference on Very large data bases*, pages 13–24. VLDB Endowment, 2005.

[39] Graham Cormode, Minos Garofalakis, S. Muthukrishnan, and Rajeev Rastogi. Holistic aggregates in a networked world: distributed tracking of approximate quantiles. In *SIGMOD '05: Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pages 25–36, New York, NY, USA, 2005. ACM Press.

[40] Graham Cormode and S. Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *J. Algorithms*, 55(1):58–75, 2005.

[41] Graham Cormode and S. Muthukrishnan. Space efficient mining of multigraph streams. In *PODS '05: Proceedings of the twenty-fourth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 271–282, New York, NY, USA, 2005. ACM Press.

[42] Graham Cormode, S. Muthukrishnan, and Wei Zhuang. What's Different: Distributed, Continuous Monitoring of Duplicate-Resilient Aggregates on Data Streams. In *ICDE '06: Proceedings of the 22nd International Conference on Data Engineering (ICDE'06)*, page 57, Washington, DC, USA, 2006. IEEE Computer Society.

[43] Manuel Costa, Miguel Castro, Antony Rowstron, and Peter Key. PIC: Practical Internet Coordinates for Distance Estimation. In *ICDCS '04: Proceedings of the 24th International Conference on Distributed Computing Systems (ICDCS'04)*, pages 178–187, Washington, DC, USA, 2004. IEEE Computer Society.

[44] Chuck Cranor, Theodore Johnson, Oliver Spataschek, and Vladislav Shkapenyuk. Gigascope: a stream database for network applications. In *SIGMOD '03: Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 647–651, New York, NY, USA, 2003. ACM Press.

[45] Frank Dabek, Russ Cox, Frans Kaashoek, and Robert Morris. Vivaldi: a decentralized network coordinate system. In *SIGCOMM '04: Proceedings of the 2004 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 15–26, New York, NY, USA, 2004. ACM Press.

[46] A. Das, S. Ganguly, M. Garofalakis, and R. Rastogi. Distributed set-expression cardinality estimation, 2004.

[47] 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.

[48] Umeshwar Dayal, Jennifer Widom, and Stefano Ceri. *Active Database Systems: Triggers and Rules for Advanced Database Processing.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1994.

[49] S.C. Deerwester, S.T. Dumais, T.K. Landauer, G.W. Furnas, and R.A. Harshman. Indexing by latent semantic analysis. *Journal of the American Society for Information Science*, 41(6):391–407, 1990.

[50] A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, and D. Terry. Epidemic algorithms for replicated database maintenance. In *Proc. 6th ACM PODC*, pages 1–12, 1987.

[51] Mark Dilman and Danny Raz. Efficient reactive monitoring. In *INFOCOM*, pages 1012–1019, 2001.

[52] P. Dini, G. v. Bochmann, T. Koch, and B. Krämer. Agent based management of distributed systems with variable polling frequency policies. In *Proceedings of the fifth IFIP/IEEE international symposium on Integrated network management V : integrated management in a virtual world*, pages 553–564, London, UK, UK, 1997. Chapman & Hall, Ltd.

[53] Danny Dolev, Yuval Harari, Nathan Linial, Noam Nisan, and Michal Parnas. Neighborhood preserving hashing and approximate queries. *SIAM Journal on Discrete Mathematics*, 15(1):73–85, 2002.

[54] John R. Douceur, Atul Adya, William J. Bolosky, Dan Simon, and Marvin Theimer. Reclaiming space from duplicate files in a serverless distributed file system. In *ICDCS '02: Proceedings of the 22 nd International Conference on Distributed Computing Systems (ICDCS'02)*, page 617, Washington, DC, USA, 2002. IEEE Computer Society.

[55] C. Faloutsos, R. Barber, M. Flickner, J. Hafner, W. Niblack, D. Petkovic, and W. Equitz. Efficient and effective querying by image content. *J. Intell. Inf. Syst.*, 3(3-4):231–262, 1994.

[56] Li Fan, Pei Cao, Jussara Almeida, and Andrei Z. Broder. Summary cache: a scalable wide-area web cache sharing protocol. *SIGCOMM Comput. Commun. Rev.*, 28(4):254–265, 1998.

[57] Philippe Flajolet and G. Nigel Martin. Probabilistic counting algorithms for data base applications. *J. Comput. Syst. Sci.*, 31(2):182–209, 1985.

[58] Lisa Fleischer, Michel X. Goemans, Vahab S. Mirrokni, and Maxim Sviridenko. Tight approximation algorithms for maximum general assignment problems. In *SODA '06: Proceedings of the seventeenth annual ACM-SIAM symposium on Discrete algorithm*, pages 611–620, New York, NY, USA, 2006. ACM Press.

[59] Chuck Fraleigh, Christophe Diot, Bryan Lyles, Sue B. Moon, Philippe Owezarski, Dina Papagiannaki, and Fouad A. Tobagi. Design and deployment of a passive monitoring infrastructure. In *IWDC '01: Proceedings of the Thyrrhenian International Workshop on Digital Communications*, pages 556–575, London, UK, 2001. Springer-Verlag.

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

[61] Deepak Ganesan, Gaurav Mathur, and Prashant J. Shenoy. Rethinking data management for storage-centric sensor networks. In *CIDR*, pages 22–31. www.crdrdb.org, 2007.

[62] Sumit Ganguly, Minos Garofalakis, and Rajeev Rastogi. Tracking set-expression cardinalities over continuous update streams. *The VLDB Journal*, 13(4):354–369, 2004.

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

[64] Shahram Ghandeharizadeh and Richard Muntz. Design and implementation of scalable continuous media servers. *Parallel Comput.*, 24(1):91–122, 1998.

[65] Phillip B. Gibbons, Yossi Matias, and Viswanath Poosala. Fast incremental maintenance of approximate histograms. *ACM Trans. Database Syst.*, 27(3):261–298, 2002.

[66] Anna C. Gilbert, Sudipto Guha, Piotr Indyk, Yannis Kotidis, S. Muthukrishnan, and Martin J. Strauss. Fast, small-space algorithms for approximate histogram maintenance. In *STOC '02: Proceedings of the thiry-fourth annual ACM symposium on Theory of computing*, pages 389–398, New York, NY, USA, 2002. ACM Press.

[67] Michel Goemans, Li Erran Li, Vahab S. Mirrokni, and Marina Thottan. Market sharing games applied to content distribution in ad-hoc networks. In *MobiHoc '04: Proceedings of the 5th ACM international symposium on Mobile ad hoc networking and computing*, pages 55–66, New York, NY, USA, 2004. ACM Press.

[68] Michel Goemans, Vahab Mirrokni, and Adrian Vetta. Sink equilibria and convergence. In *FOCS '05: Proceedings of the 46th Annual IEEE Symposium on Foundations of Computer Science*, pages 142–154, Washington, DC, USA, 2005. IEEE Computer Society.

[69] M.K. Goldberg. Edge-Coloring of multigraphs: Recoloring technique. J. Graph Theory, 8:121-137, 1984.

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

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

[72] Vijay Gopalakrishnan, Bujor Silaghi, Bobby Bhattacharjee, and Pete Keleher. Adaptive replication in peer-to-peer systems. In *The 24th International Conference on Distributed Computing Systems*, March 2004.

[73] Michael Greenwald and Sanjeev Khanna. Space-efficient online computation of quantile summaries. In *SIGMOD '01: Proceedings of the 2001 ACM SIGMOD international conference on Management of data*, pages 58–66, New York, NY, USA, 2001. ACM Press.

[74] Michael B. Greenwald and Sanjeev Khanna. Power-conserving computation of order-statistics over sensor networks. In *PODS '04: Proceedings of the twenty-third ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 275–285, New York, NY, USA, 2004. ACM Press.

[75] Sudipto Guha, Andrew McGregor, and Suresh Venkatasubramanian. Streaming and sublinear approximation of entropy and information distances. In *SODA '06: Proceedings of the seventeenth annual ACM-SIAM symposium on Discrete algorithm*, pages 733–742, New York, NY, USA, 2006. ACM Press.

[76] Sudipto Guha and Kamesh Munagala. Improved algorithms for the data placement problem. In *SODA '02: Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 106–107, Philadelphia, PA, USA, 2002. Society for Industrial and Applied Mathematics.

[77] A. Gupta, D. Agrawal, and A. El Abbadi. Approximate range selection queries in peer-to-peer systems. In *Proceedings of the First Biennial Conference on Innovative Data Systems Research*, Asilomar, California, United States, January 2003.

[78] A. Gupta, B. Liskov, and R. Rodrigues. Efficient routing for peer-to-peer overlays. In *Proc. First Symposium on Networked Systems Design and Implementation (NSDI '04)*, March 2004.

[79] Indranil Gupta, Ken Birman, Prakash Linga, Al Demers, and Robbert van Renesse. Kelips: Building an efficient and stable P2P DHT through increased memory and background overhead. In *Proceedings of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS '03)*, 2003.

[80] Indranil Gupta, Robbert van Renesse, and Kenneth P. Birman. Scalable fault-tolerant aggregation in large process groups. In *DSN '01: Proceedings of the 2001 International Conference on Dependable Systems and Networks (formerly: FTCS)*, pages 433–442, Washington, DC, USA, 2001. IEEE Computer Society.

[81] Marios Hadjieleftheriou, John W. Byers, and George Kollios. Robust sketching and aggregation of distributed data streams. technical report 2005-11, boston university computer science department, 2005.

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

[83] Nicholas Harvey, Michael B. Jones, Stefan Saroiu, Marvin Theimer, and Alec Wolman. Skipnet: A scalable overlay network with practical locality properties. In *In proceedings of the 4th USENIX Symposium on Internet Technologies and Systems (USITS '03)*, Seattle, WA, March 2003.

[84] Dorit S Hochbaum, Takao Nishizeki, and David B Shmoys. A better than best possible algorithm to edge color multigraphs. *J. Algorithms*, 7(1):79–104, 1986.

[85] L. Huang, X. Nguyen, M. Garofalakis, J. Hellerstein, M. Jordan, A. Joseph, and Nina Taft. Distributed PCA and Network Anomaly Detection. In *In Proceedings of INFOCOM 2007.*, 2007.

[86] L. Huang, X. Nguyen, M. Garofalakis, M. Jordan, A. Joseph, and N. Taft. Distributed PCA and Network Anomaly Detection. In *In Proceedings of NIPS 2006.*, 2006.

[87] Ling Huang, Minos Garofalakis, Joseph Hellerstein, Anthony Joseph, and Nina Taft. Toward sophisticated detection with distributed triggers. In *MineNet '06: Proceedings of the 2006 SIGCOMM workshop on Mining network data*, pages 311–316, New York, NY, USA, 2006. ACM Press.

[88] www.ibm.com/software/tivoli/products/monitor/.

[89] Piotr Indyk, Rajeev Motwani, Prabhakar Raghavan, and Santosh Vempala. Locality-preserving hashing in multidimensional spaces. In *STOC '97: Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, pages 618–625, New York, NY, USA, 1997. ACM Press.

[90] Intel-Dante Monitoring Project. http://www.cambridge.intel-research.net/monitoring/dante/.

[91] Ankur Jain, Joseph M. Hellerstein, Sylvia Ratnasamy, and David Wetherall. A wakeup call for internet monitoring systems: The case for distributed triggers. In *Proc. 3rd ACM SIGCOMM Workshop on Hot Topics in Networks (HotNets)*, San Diego, CA, November 2004.

[92] Navendu Jain, Praveen Yalagandula, Mike Dahlin, and Yin Zhang. INSIGHT: a distributed monitoring system for tracking continuous queries. In *SOSP '05: Proceedings of the twentieth ACM symposium on Operating systems principles - Work in progress session*, pages 1–7, New York, NY, USA, 2005. ACM Press.

[93] Márk Jelasity and Alberto Montresor. Epidemic-style proactive aggregation in large overlay networks. In *ICDCS '04: Proceedings of the 24th International Conference on Distributed Computing Systems (ICDCS'04)*, pages 102–109, Washington, DC, USA, 2004. IEEE Computer Society.

[94] J. Jiao, S. Naqvi, D. Raz, and B. Sugla. Toward efficient monitoring. IEEE Journal on Selected Areas in Communications, 18(5):723–732, May 2000., 2000.

[95] David S. Johnson, Christos H. Papadimtriou, and Mihalis Yannakakis. How easy is local search? *J. Comput. Syst. Sci.*, 37(1):79–100, 1988.

[96] Theodore Johnson, S. Muthukrishnan, Vladislav Shkapenyuk, and Oliver Spatscheck. A heartbeat mechanism and its application in gigascope. In *VLDB '05: Proceedings of the 31st international conference on Very large data bases*, pages 1079–1088. VLDB Endowment, 2005.

[97] A. D. Joseph, L. Huang, M. Garofalakis, and N. Taft. Communication-efficient tracking of distributed cumulative triggers. In *In Proceedings of ICDCS 2007.*, 2007.

[98] J. Jung, B. Krishnamurthy, and M. Rabinovich. Flash crowds and denial of service attacks: Characterization and implications for CDNs and web sites. In Proceedings of the International World Wide Web Conference, pages 252–262. IEEE, May 2002.

[99] M. Frans Kaashoek and David R. Karger. Koorde: A simple degree-optimal distributed hash table. In *Proceedings of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS '03)*, 2003.

[100] David Karger, Eric Lehman, Tom Leighton, Mathhew Levine, Daniel Lewin, and Rina Panigrahy. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *ACM Symposium on Theory of Computing*, pages 654–663, May 1997.

[101] R. Karp, C. Schindelhauer, S. Shenker, and B. Vocking. Randomized rumor spreading. In *Proc. 41st IEEE FOCS*, pages 565–574, 2000.

[102] S. Kashyap and S. Khuller. Experimental evaluation of data placement algorithms. *Manuscript*, 2002.

[103] S. Kashyap, J. Ramamirtham, R. Rastogi, and P. Shukla. Efficient constraint monitoring using adaptive thresholds. Technical Report ITD-06-47318H, Bell Labs Technical Memorandum, 2006.

[104] Srinivas Kashyap, Supratim Deb, K. V. M. Naidu, Rajeev Rastogi, and Anand Srinivasan. Efficient gossip-based aggregate computation. In *PODS '06: Proceedings of the twenty-fifth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 308–317, New York, NY, USA, 2006. ACM Press.

[105] Srinivas Kashyap and Samir Khuller. Algorithms for non-uniform size data placement on parallel disks. *J. Algorithms*, 60(2):144–167, 2006.

[106] Srinivas Kashyap, Samir Khuller, Yung-Chun Wan, and Leana Golubchik. Fast algorithms for data reconfiguration in parallel disks. In *Proceedings of the Eighth Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 95–107. ACM and SIAM, 2006.

[107] David Kempe, Alin Dobra, and Johannes Gehrke. Gossip-based computation of aggregate information. In *FOCS '03: Proceedings of the 44th Annual IEEE Symposium on Foundations of Computer Science*, page 482, Washington, DC, USA, 2003. IEEE Computer Society.

[108] Ram Keralapura, Graham Cormode, and Jeyashankher Ramamirtham. Communication-efficient distributed monitoring of thresholded counts. In *SIGMOD '06: Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 289–300, New York, NY, USA, 2006. ACM Press.

[109] Samir Khuller, Yoo-Ah Kim, and Yung-Chun (Justin) Wan. Algorithms for data migration with cloning. In *PODS '03: Proceedings of the twenty-second ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 27–36, New York, NY, USA, 2003. ACM Press.

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

[111] Bong-Jun Ko and Dan Rubenstein. Distributed self-stabilizing placement of replicated resources in emerging networks. *IEEE/ACM Trans. Netw.*, 13(3):476–487, 2005.

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

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

[114] Lintao Liu and Kang-Won Lee. Keyword fusion to support efficient keyword-based search in peer-to-peer file sharing. In *CCGRID '04: Proceedings of the 2004 IEEE International Symposium on Cluster Computing and the Grid*, pages 269–276, Washington, DC, USA, 2004. IEEE Computer Society.

[115] Qin Lv, Pei Cao, Edith Cohen, Kai Li, and Scott Shenker. Search and replication in unstructured peer-to-peer networks. In *ICS '02: Proceedings of the 16th international conference on Supercomputing*, pages 84–95, New York, NY, USA, 2002. ACM Press.

[116] Samuel Madden, Michael J. Franklin, Joseph M. Hellerstein, and Wei Hong. TAG: a Tiny AGgregation service for ad-hoc sensor networks. *SIGOPS Oper. Syst. Rev.*, 36(SI):131–146, 2002.

[117] Samuel Madden, Michael J. Franklin, Joseph M. Hellerstein, and Wei Hong. The design of an acquisitional query processor for sensor networks. In *SIGMOD '03: Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 491–502, New York, NY, USA, 2003. ACM Press.

[118] Dahlia Malkhi, Moni Naor, and David Ratajczak. Viceroy: a scalable and dynamic emulation of the butterfly. In *PODC '02: Proceedings of the twenty-first annual symposium on Principles of distributed computing*, pages 183–192, New York, NY, USA, 2002. ACM Press.

[119] Amit Manjhi, Suman Nath, and Phillip B. Gibbons. Tributaries and deltas: efficient and robust aggregation in sensor network streams. In *SIGMOD '05: Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pages 287–298, New York, NY, USA, 2005. ACM Press.

[120] Amit Manjhi, Vladislav Shkapenyuk, Kedar Dhamdhere, and Christopher Olston. Finding (Recently) Frequent Items in Distributed Data Streams. In *ICDE '05: Proceedings of the 21st International Conference on Data Engineering (ICDE'05)*, pages 767–778, Washington, DC, USA, 2005. IEEE Computer Society.

[121] Petar Maymounkov and David Mazières. Kademlia: A Peer-to-Peer Information System Based on the XOR Metric. In *IPTPS '01: Revised Papers from the First International Workshop on Peer-to-Peer Systems*, pages 53–65, London, UK, 2002. Springer-Verlag.

[122] Adam Meyerson, Kamesh Munagala, and Serge Plotkin. Web caching using access statistics. In *SODA '01: Proceedings of the twelfth annual ACM-SIAM symposium on Discrete algorithms*, pages 354–363, Philadelphia, PA, USA, 2001. Society for Industrial and Applied Mathematics.

[123] Michael Mitzenmacher. Compressed bloom filters. In *Proceedings of the twentieth annual ACM symposium on Principles of distributed computing*, pages 144–150. ACM Press, 2001.

[124] Alper Tugay Mizrak, Yuchung Cheng, Vineet Kumar, and Stefan Savage. Structured superpeers: Leveraging heterogeneity to provide constant-time lookup. In *Proceedings of the Third IEEE Workshop on Internet Applications (WIAPP'03)*, pages 104–111, San Jose, California, june 2003.

[125] R. Motwani and P. Raghavan. *Randomized Algorithms.* Cambridge University Press, USA, 1995.

[126] Kyriakos Mouratidis, Spiridon Bakiras, and Dimitris Papadias. Continuous monitoring of top-k queries over sliding windows. In *SIGMOD '06: Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 635–646, New York, NY, USA, 2006. ACM Press.

[127] How much information? School of Information Management and Systems. University of California at Berkeley. http://www.sims.berkeley.edu/research/projects/how-much-info 2003/.

[128] Moni Naor and Udi Wieder. Novel architectures for P2P applications: the continuous-discrete approach. In *SPAA '03: Proceedings of the fifteenth annual ACM symposium on Parallel algorithms and architectures*, pages 50–59, New York, NY, USA, 2003. ACM Press.

[129] Suman Nath, Phillip B. Gibbons, Srinivasan Seshan, and Zachary R. Anderson. Synopsis diffusion for robust aggregation in sensor networks. In *SenSys '04: Proceedings of the 2nd international conference on Embedded networked sensor systems*, pages 250–262, New York, NY, USA, 2004. ACM Press.

[130] Suman Nath and Aman Kansal. FlashDB: dynamic self-tuning database for NAND flash. In *IPSN '07: Proceedings of the 6th international conference on Information processing in sensor networks*, pages 410–419, New York, NY, USA, 2007. ACM Press.

[131] T.S.E. Ng and Hui Zhang. Predicting internet network distance with coordinates-based approaches. In *INFOCOMM 2002. Twenty-First Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, pages 170–179, 2002.

[132] T.S.E. Ng and Hui Zhang. A network positioning system for the internet. In *USENIX 2004 Annual Technical Conference*, pages 141–154, 2004.

[133] National Laboratory for Applied Network Research. http://www.nlanr.net/.

[134] The Ganglia Distributed Monitoring System. http://ganglia.sourceforge.net/.

[135] Chris Olston, Jing Jiang, and Jennifer Widom. Adaptive filters for continuous queries over distributed data streams. In *SIGMOD '03: Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 563–574, New York, NY, USA, 2003. ACM Press.

[136] Chris Olston, Boon Thau Loo, and Jennifer Widom. Adaptive precision setting for cached approximate values. In *SIGMOD '01: Proceedings of the 2001 ACM SIGMOD international conference on Management of data*, pages 355–366, New York, NY, USA, 2001. ACM Press.

[137] C. H. Papadimitriou, A. A. Schäffer, and M. Yannakakis. On the complexity of local search. In *STOC '90: Proceedings of the twenty-second annual ACM symposium on Theory of computing*, pages 438–445, New York, NY, USA, 1990. ACM Press.

[138] Alexander P. Pentland, Rosalind W. Picard, and S. Scarloff. Photobook: tools for content-based manipulation of image databases. volume 2185, pages 34–47. SPIE, 1994.

[139] M. Pias, J. Crowcroft, S. Wilbur, S. Bhatti, and T. Harris. Lighthouses for scalable distributed location. In Second International Workshop on Peer-to-Peer Systems (IPTPS '03), Feb 2003.

[140] B. Pittel. On spreading a rumor. *SIAM J. Applied Math.*, 47(1):213–223, February 1987.

[141] C. Greg Plaxton, Rajmohan Rajaraman, and Andréa W. Richa. Accessing nearby copies of replicated objects in a distributed environment. In *SPAA '97: Proceedings of the ninth annual ACM symposium on Parallel algorithms and architectures*, pages 311–320, New York, NY, USA, 1997. ACM Press.

[142] G. J. Pottie and W. J. Kaiser. Wireless integrated network sensors. *Commun. ACM*, 43(5):51–58, 2000.

[143] Prabhakar Raghavan. Probabilistic construction of deterministic algorithms: approximating packing integer programs. *J. Comput. Syst. Sci.*, 37(2):130–143, 1988.

[144] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Schenker. A scalable content-addressable network. In *SIGCOMM '01: Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 161–172, New York, NY, USA, 2001. ACM Press.

[145] D. Raz and Y. Shavitt. Toward efficient distributed network management. *J. Netw. Syst. Manage.*, 9(3):347–361, 2001.

[146] Robbert Van Renesse, Kenneth P. Birman, and Werner Vogels. Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining. *ACM Trans. Comput. Syst.*, 21(2):164–206, 2003.

[147] Patrick Reynolds and Amin Vahdat. Efficient peer-to-peer keyword searching. In *Proceedings of International Middleware Conference*, pages 21–40, jun 2003.

[148] Sean Rhea, Dennis Geels, Timothy Roscoe, and John Kubiatowicz. Handling churn in a DHT. In *ATEC'04: Proceedings of the USENIX Annual Technical Conference 2004 on USENIX Annual Technical Conference*, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association.

[149] Antony I. T. Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Middleware '01: Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg*, pages 329–350, London, UK, 2001. Springer-Verlag.

[150] Alejandro A. Schäffer and Mihalis Yannakakis. Simple local search problems that are hard to solve. *SIAM J. Comput.*, 20(1):56–87, 1991.

[151] Christian Scheideler. How to spread adversarial nodes?: rotate! In *STOC '05: Proceedings of the thirty-seventh annual ACM symposium on Theory of computing*, pages 704–713, New York, NY, USA, 2005. ACM Press.

[152] Christian Schindelhauer and Gunnar Schomaker. Weighted distributed hash tables. In *SPAA '05: Proceedings of the seventeenth annual ACM symposium on Parallelism in algorithms and architectures*, pages 218–227, New York, NY, USA, 2005. ACM Press.

[153] Cristina Schmidt and Manish Parashar. Flexible information discovery in decentralized distributed systems. In *HPDC '03: Proceedings of the 12th IEEE International Symposium on High Performance Distributed Computing (HPDC'03)*, page 226, Washington, DC, USA, 2003. IEEE Computer Society.

[154] Beomjoo Seo and Roger Zimmermann. Efficient disk replacement and data migration algorithms for large disk subsystems. *Trans. Storage*, 1(3):316–345, 2005.

[155] H. Shachnai and T. Tamir. Polynomial time approximation schemes for class-constrained packing problems. In *Proceedings of Workshop on Approximation Algorithms (APPROX). LNCS 1913*, pages 238–249. Springer-Verlag, 2000.

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

[157] H. Shachnai and T. Tamir. Approximation schemes for generalized 2-dimensional vector packing with application to data placement. In *Proceedings of Workshop on Approximation Algorithms (APPROX). LNCS 2764*, pages 165–177. Springer-Verlag, 2003.

[158] Izchak Sharfman, Assaf Schuster, and Daniel Keren. A geometric approach to monitoring threshold functions over distributed data streams. In *SIGMOD '06: Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 301–312, New York, NY, USA, 2006. ACM Press.

[159] Y. Shavitt and T. Tankel. Big-bang simulation for embedding network distances in euclidean space. In *INFOCOM 2003. Twenty-Second Annual Joint Conference of the IEEE Computer and Communications Societies. IEEE*, pages 1922–1932, 2003.

[160] Scott Shenker, Sylvia Ratnasamy, Brad Karp, Ramesh Govindan, and Deborah Estrin. Data-centric storage in sensornets. *SIGCOMM Comput. Commun. Rev.*, 33(1):137–142, 2003.

[161] S. Shi, G. Yang, D. Wang, J. Yu, S. Qu, and M. Chen. Making Peer-to-Peer Keyword Searching Feasible Using Multi-level Partitioning, in IPTPS'04, San Diego, CA, USA, February 2004., 2004.

[162] Adam Silberstein, Kamesh Munagala, and Jun Yang. Energy-efficient monitoring of extreme values in sensor networks. In *SIGMOD '06: Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 169–180, New York, NY, USA, 2006. ACM Press.

[163] Volker Stemann. Parallel balanced allocations. In *SPAA '96: Proceedings of the eighth annual ACM symposium on Parallel algorithms and architectures*, pages 261–269, New York, NY, USA, 1996. ACM Press.

[164] Ion Stoica, Robert Morris, David Liben-Nowell, David R. Karger, M. Frans Kaashoek, Frank Dabek, and Hari Balakrishnan. Chord: a scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM Trans. Netw.*, 11(1):17–32, 2003.

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

[166] C. Swamy. Unpublished Manuscript. To be combined with the paper Approximation algorithms for data placement in arbitrary networks from SODA 2001.

[167] Chunqiang Tang, Zhichen Xu, and Sandhya Dwarkadas. Peer-to-peer information retrieval using self-organizing semantic overlay networks. In *SIGCOMM '03: Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 175–186, New York, NY, USA, 2003. ACM Press.

[168] L. Tang and M. Crovella. Virtual landmarks for the Internet. In Internet Measurement Conference, pages 143-152, Miami Beach, FL, October 2003.

[169] Nitin Thaper, Sudipto Guha, Piotr Indyk, and Nick Koudas. Dynamic multidimensional histograms. In *SIGMOD '02: Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 428–439, New York, NY, USA, 2002. ACM Press.

[170] Introduction to Storage Area Networks. IBM Redbook. http://www.redbooks.ibm.com/.

[171] John Wilkes, Richard Golding, Carl Staelin, and Tim Sullivan. The HP AutoRAID hierarchical storage system. *ACM Trans. Comput. Syst.*, 14(1):108–136, 1996.

[172] Joel L. Wolf, Philip S. Yu, and Hadas Shachnai. DASD dancing: a disk load balancing optimization scheme for video-on-demand computer systems. *SIGMETRICS Perform. Eval. Rev.*, 23(1):157–166, 1995.

[173] Ouri Wolfson, Sushil Jajodia, and Yixiu Huang. An adaptive data replication algorithm. *ACM Trans. Database Syst.*, 22(2):255–314, 1997.

[174] Bernard Wong, Aleksandrs Slivkins, and Emin Gün Sirer. Meridian: a lightweight network location service without virtual coordinates. *SIGCOMM Comput. Commun. Rev.*, 35(4):85–96, 2005.

[175] Alec Woo, Sam Madden, and Ramesh Govindan. Networking support for query processing in sensor networks. *Commun. ACM*, 47(6):47–52, 2004.

[176] Praveen Yalagandula and Mike Dahlin. A scalable distributed information management system. In *SIGCOMM '04: Proceedings of the 2004 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 379–390, New York, NY, USA, 2004. ACM Press.

[177] Praveen Yalagandula, Puneet Sharma, Sujata Banerjee, Sujoy Basu, and Sung-Ju Lee. S3: a scalable sensing service for monitoring large networked systems. In *INM '06: Proceedings of the 2006 SIGCOMM workshop on Internet network management*, pages 71–76, New York, NY, USA, 2006. ACM Press.

[178] Y. Yao and J. Gehrke. Query processing in sensor networks. In *Proc. 1st CIDR*, 2003.

[179] B. Y. Zhao, J. D. Kubiatowicz, and A. D. Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Technical Report UCB/CSD-01-1141, UC Berkeley, April 2001.