Generalized Optimal Response Time Retrieval of Replicated Data from Storage Arrays

NIHAT ALTIPARMAK and ALI ŞAMAN TOSUN, University of Texas at San Antonio

5

Declustering techniques reduce query response times through parallel I/O by distributing data among parallel disks. Recently, replication-based approaches were proposed to further reduce the response time. Efficient retrieval of replicated data from multiple disks is a challenging problem. Existing retrieval techniques are designed for storage arrays with identical disks, having no initial load or network delay. In this article, we consider the generalized retrieval problem of replicated data where the disks in the system might be heterogeneous, the disks may have initial load, and the storage arrays might be located on different sites. We first formulate the generalized retrieval problem using a Linear Programming (LP) model and solve it with mixed integer programming techniques. Next, the generalized retrieval problem is formulated as a more efficient maximum flow problem. We prove that the retrieval schedule returned by the maximum flow technique yields the optimal response time and this result matches the LP solution. We also propose a low-complexity online algorithm for the generalized retrieval problem by not guaranteeing the optimality of the result. Performance of proposed and state of the art retrieval strategies are investigated using various replication schemes, query types, query loads, disk specifications, network delays, and initial loads.

Categories and Subject Descriptors: D.4.2 [**Operating Systems**]: Storage Management

General Terms: Design, Algorithms, Performance

Additional Key Words and Phrases: Declustering, replication, storage arrays, generalized retrieval, maximum flow, linear programming

ACM Reference Format:
Altiparmak, N. and Tosun, A. Ş. 2013. Generalized optimal response time retrieval of replicated data from storage arrays. *ACM Trans. Storage* 9, 2, Article 5 (July 2013), 36 pages.
DOI: http://dx.doi.org/10.1145/2491472.2491474

1. INTRODUCTION

Traditional retrieval methods based on index structures developed for single disk and single processor environments [Beckmann et al. 1990; Gaede and Gunther 1998; Guttman 1984; Samet 1989] are ineffective for the storage and retrieval in multiple processor and multiple disk environments. Since the amount of data is large, it is very natural to use parallel disk architectures in these systems. Besides scalability with respect to storage, storage arrays offer the opportunity to exploit I/O parallelism during retrieval. The most crucial part of exploiting I/O parallelism is to develop storage techniques that access the data in parallel. *Declustering* is the most common approach for efficient parallel I/O. The data space is partitioned into disjoint regions (buckets),

This research was supported by US National Science Foundation (NSF) Grant CNS-0855247.
An earlier version of this article appeared in *Proceedings of the 28th International Conference on Distributed Computing Systems (ICDCS'08)* [Tosun 2008].
Authors' address: N. Altiparmak and A. Ş. Tosun, Computer Science Department, University of Texas at San Antonio; email: {naltipar, tosun}@cs.utsa.edu.

DOI: http://dx.doi.org/10.1145/2491472.2491474

and data is allocated to multiple disks. When users issue a query, data falling into disjoint partitions is retrieved in parallel from multiple disks.

After allocating the data to the storage array efficiently, the next step is to retrieve the buckets of a given query so that the response time of the query is minimum. The solution is trivial if the buckets of the query are not replicated across multiple disks, since without replication, there is only one candidate disk from which a bucket can be retrieved. However, replicated declustering makes the optimal response time retrieval problem more complicated. Maximum flow is the general technique used in the optimal response time retrieval of replicated data. However, existing retrieval techniques are designed for storage arrays with identical disks. Recently, hybrid storage arrays consisting of solid-state and rotating disks have appeared on the market. Beside the heterogeneity of the disks, current retrieval techniques also assume that replication is done at a single site using disks with no initial load or network delay. Optimal response time retrieval of the replicated data from heterogeneous storage systems located on multiple sites with initial load and network delay remains an open problem.

Optimal response time retrieval problem of replicated data was first formulated by Chen and Rotem [1994] along with a maximum flow solution for homogeneous storage arrays located on a single site. Tosun [2008] extended the problem by locating the storage arrays on two different sites connected with a dedicated network. A maximum flow solution including an online algorithm is proposed for this extended problem. This article extends Tosun [2008] further by defining the generalized optimal response time retrieval problem that handles heterogeneous storage systems, disks with initial loads, and any number of sites having different network delays. We propose two fundamentally different approaches to the generalized problem. Our first approach uses the LP model and the second uses the maximum flow formulation. We theoretically prove that the retrieval schedule returned by the maximum flow approach yields the optimal response time and we observe that this result matches the optimal response time calculated by the LP approach. A low complexity online algorithm is also proposed for the generalized retrieval problem of not guaranteeing the optimality of the result. Extensive experiments are performed using different parameters of the generalized retrieval problem in order to compare the performances of proposed and state of the art retrieval strategies.

The rest of the article is organized as follows. In Section 2 we present the background information and the related work on declustering. Section 3 explains our application model. Section 4 describes the basic retrieval problem and Section 5 provides the generalized retrieval problem along with the proposed solutions. We investigate the performance of the proposed algorithms in Section 6 and conclude with Section 7.

2. BACKGROUND AND RELATED WORK

In this section, we provide the preliminaries and the related work on declustering.

2.1. Preliminaries

For a homogeneous storage array with N disks, an allocation policy is said to be *strictly optimal* if no query Q that retrieves $|Q|$ buckets, has more than $\lceil \frac{|Q|}{N} \rceil$ buckets allocated to the same disk. The most common query type is a *range query*. In a *range query*, the user specifies an area of interest using a range of values for each dimension. The result of the *range query* is the set of buckets in the dataset that have values within the specified range for each dimension. Except a few restricted cases, it is impossible to reach strict optimality for spatial range queries [Abdel-Ghaffar and El Abbadi 1997]. The lower bound on extra disk accesses is logarithmic in N [Bhatia et al. 2000].

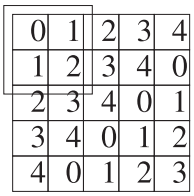


Fig. 1. Declustering of 5×5 grid.

A declustering of a 5×5 grid using 5 disks is given in Figure 1. Each square denotes a bucket and the number on the square denotes the disk that the bucket is stored at. An $i \times j$ range query has i rows and j columns. For retrieval of an $i \times j$ range query, the best we can expect is $\lceil \frac{i*j}{5} \rceil$ and this happens if the buckets of the query are spread to the disks in a balanced way. In most cases, this is not possible. We use the notation $[i,j], 0 \leq i,j \leq N-1$ to denote the bucket in row i and column j. A query can be represented as a set using this notation. Consider the 2×2 query $Q_1 = \{[0,0],[0,1],[1,0],[1,1]\}$ shown in Figure 1. Since 2 buckets of the query are stored on disk 1 it requires 2 disk accesses. Deviation from optimal retrieval cost $\lceil \frac{i*j}{5} \rceil$ is called additive error. For the 2×2 query the additive error is 1. Additive error of a scheme is the maximum additive error over all the queries.

2.2. Related Work

Several methods have been proposed for declustering data, including Disk Modulo [Du and Sobolewski 1982], Field-wise Exclusive OR [Kim and Pramanik 1988], Hilbert [Faloutsos and Bhagwat 1993], General Multidimensional Data Allocation [Hua and Young 1997], cyclic allocation schemes [Prabhakar et al. 1998a, 1998b], Golden Ratio Sequences [Chen et al. 2000], Hierarchical [Bhatia et al. 2000], Discrepancy declustering [Chen and Cheng 2002], and Threshold-Based Declustering [Tosun 2005a, 2005c, 2007b]. Some declustering techniques utilize information about query distribution [Ghandeharizadeh and DeWitt 1990a, 1990b]. Use of combinatorial designs including Latin squares [Kim and Prasanna-Kumar 1993] and Latin cubes [Fan et al. 1994] is proposed for a variant of declustering problems where array blocks are distributed among multiple memory modules. When the number of disks is a power of two, a declustering scheme that achieves the lower bound is proposed by Atallah and Prabhakar [2000]. Optimization-based approaches [Koyuturk and Aykanat 2005; Liu and Wu 2001; Shektar and Liu 1996] are proposed to handle arbitrary datasets and queries.

All of these declustering schemes were designed assuming a single copy of the data. Recently, replication strategies for spatial range queries [Chen and Cheng 2003; Ferhatosmanoglu et al. 2004; Frikken 2005; Frikken et al. 2002] and arbitrary queries [Oktay et al. 2009; Tosun 2004, 2005b] were proposed. Replication improves the worst-case additive error for declustering using multiple copies of the data. In addition to offering lower worst-case additive error, replication has many other advantages including better fault-tolerance and support for queries of arbitrary shape. Readers are directed to Tosun [2007a] for an in-depth comparison and analysis of replicated declustering schemes.

3. APPLICATION MODEL

Many applications have data generated at multiple sites and queried by users from multiple sites. Storing all the data at a central site is impractical. Replication of data at multiple storage arrays at distant locations is necessary since location-based server

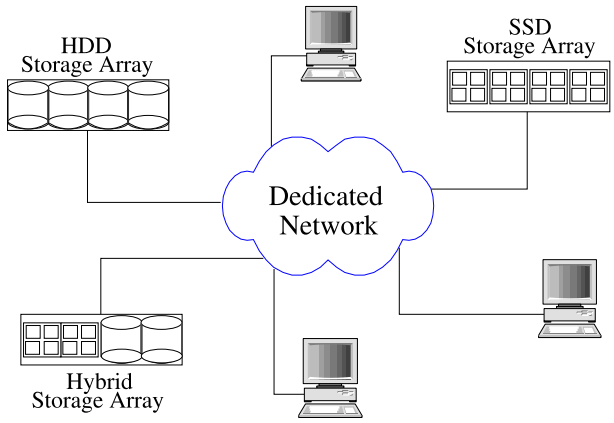


Fig. 2. Storage arrays with a dedicated network.

selection can be used and the load can be distributed among the servers. An example model is provided in Figure 2, where geographically distant storage arrays are connected over a dedicated network.

Many Internet service providers now offer dedicated Internet access with bandwidth, latency, packet loss, and availability guarantees. For example, XO communications' dedicated Internet access [XO] has the following guarantees. Round-trip latency of 65 milliseconds edge-to-edge within the XO network. Round-trip packet loss of at most 1% edge-to-edge within the XO network and availability of 100%. All the guarantees are measured over a calendar month.

Although traditional HDD (hard disk drive)-based storage arrays still dominate the market share, SSD (solid-state drive)-based storage arrays [Nimbus 2010; Ramsan 2010; Sun 2009b; Violin 2010, 2011] and hybrid storage arrays [Adaptec 2010; EqualLogic 2011; Sun 2009a; Zebi 2012] composed of SSDs and HDDs have gained a lot of attention recently. Our model assumes that the storage arrays can be HDD based, SSD based, or hybrid.

Potential applications of the model include the following.

— Dataset for an application is stored on a storage array. A new high-end storage array is purchased. Instead of moving all the data to the new storage array, a system spanning the two storage arrays can be used. Storage arrays are costly and making the most out of them is crucial.
— A large dataset is split and stored at storage arrays at multiple sites. We want to run an application that potentially processes parts of the whole dataset. The model allows us to do this efficiently.
— An SSD based or hybrid storage array is added to a storage system. Since SSDs have write limitations, it is necessary to use them together with other storage arrays and the model above can be used for this purpose.
— High-end computer centers with storage arrays can be combined to create storage systems with much larger capacity in an affordable way.

4. BASIC RETRIEVAL PROBLEM

In the optimal response time retrieval problem, we have N disks and $|Q|$ buckets. Each bucket can be replicated among multiple disks. The goal is to find a way of retrieving the requested buckets of a query from the disks so that the overall response time of the

Table I. Notation

Notation	Meaning
N	Total number of disks in the system
S	Total number of sites
$\|Q\|$	Total number of buckets to be retrieved; query size
c	Number of copies for each bucket
B_{ij}	1 if bucket i is retrieved from disk j, 0 otherwise
L_j	Total number of buckets retrieved from disk j; $\sum_{i=1}^{\|Q\|} B_{ij}$
I_j	0 when L_j is zero, 1 when $L_j > 0$
C_j	Average retrieval cost of a single bucket from disk j
X_j	Time it takes for disk j to process initial load before new requests can be handled
D_j	Network delay to the site where disk j is located
R	Optimal response time

query is minimized. For the basic problem, we assume that the disks are homogeneous without having any initial load or network delay and they are all located on a single site. In this case, the overall response time of the query is determined by the disk that is used to retrieve the maximum amount of buckets. In other words, we need to retrieve as few buckets as possible from the disk that is used to retrieve the maximum amount of buckets.

The notation used in this article along with the meaning is provided in Table I.

4.1. LP Solution

The basic problem can be formulated in LP as follows. For a given query, there are N different cost functions r_j that measure the retrieval cost of a query for each disk j.

$$r_j = \sum_{i=1}^{|Q|} B_{ij}, \quad j = 1, \ldots, N. \tag{1}$$

In other words, r_j holds the number of buckets L_j that are retrieved from disk j. We want to

$$\begin{aligned} \textit{Minimize}: \quad & \textit{Maximum}\{r_1, r_2, \ldots, r_N\} \\ \textit{Subject to}: \quad & r_j = \sum_{i=1}^{|Q|} B_{ij}, \quad j = 1, \ldots, N \\ & \sum_{j=1}^{N} B_{ij} = 1, \quad i = 1, \ldots, |Q|. \end{aligned} \tag{2}$$

Our second constraint, $\sum_{j=1}^{N} B_{ij} = 1, i = 1, \ldots, |Q|$, ensures that every bucket i in the query is only retrieved from a single disk. Since the formulation includes both minimization and maximization, the problem as defined, is not a linear program. However, we can convert this problem into a linear form by defining a new variable $R \geq r_j, j = 1, \ldots, N$, and minimizing R. We may rewrite this as

$$R - \sum_{i=1}^{|Q|} B_{ij} \geq 0, \quad j = 1, \ldots, N. \tag{3}$$

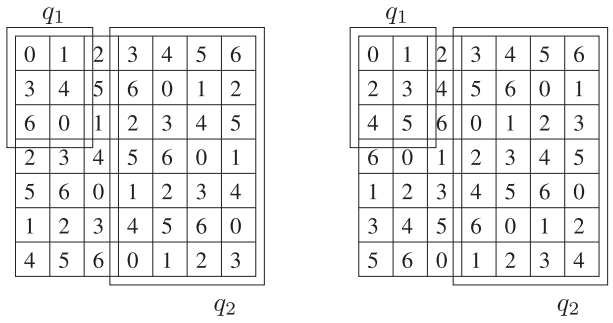


Fig. 3. Orthogonal allocation.

Thus, we can reformulate (2) as the linear program

$$\begin{aligned} &Minimize : R \\ &subject\ to : R - \sum_{i=1}^{|Q|} B_{ij} \geq 0,\ \ j = 1, \ldots, N \\ &\qquad\qquad \sum_{j=1}^{N} B_{ij} = 1, \quad i = 1, \ldots, |Q|. \end{aligned} \tag{4}$$

There are $c * |Q| + 1$ unique variables in this formulation. Although it seems like we need $N * |Q|$ variables for the B_{ij}s, since we cannot retrieve a block from a disk that it is not stored at, the number of variables required for B_{ij}s decreases to $c * |Q|$. We need an additional variable R to be used as the objective value. r_js are redefinitions of current variables. Besides the variables, we use a total of $N + |Q|$ constraints in the formulation. N comes from the first constraint of the formulation and $|Q|$ comes from the second. In LP literature, this problem is defined as minimizing the maximum of linear functions [Dantzig and Thapa 1997], and to the best of our knowledge, this is the first LP formulation for the optimal response time retrieval of replicated data.

4.2. Max-Flow Solution

The basic problem can also be solved as a max-flow problem using graph theory. When replication is used, each bucket is stored on multiple disks and we have to choose one of the disks for retrieval of the bucket. Consider the query q_1 given in Figure 3. Allocations on the left and right show the first and second copies of the data respectively. Query q_1 is a 3×2 query with optimal retrieval cost of $\lceil \frac{3\times 2}{7} \rceil = 1$. However, since in the first copy the buckets $[0, 0]$ and $[2, 1]$ are both stored on disk 0, retrieval using the first copy requires 2 disk accesses. When we consider both copies, we can represent the problem as a maximum flow problem [Chen and Rotem 1994].

For each bucket and for each disk, we create a vertex. In addition, two more vertices called source and sink are created. The source vertex s is connected to all the vertices denoting the buckets, and all the vertices denoting the disks are connected to the sink vertex t. An edge is created between vertex v_i denoting bucket i and vertex v_j denoting disk j if bucket i is stored on disk j. The next step is to define the capacities of the edges. All the edges except the ones between the disks and the sink have capacity 1. The capacity of the edges between the disks and the sink is set to $\lceil \frac{|Q|}{N} \rceil$.

The Maximum flow representation of query q_1 for a single site is given in Figure 4. Maximum flow is shown using thick lines in the figure. Since query q_1 has only 6

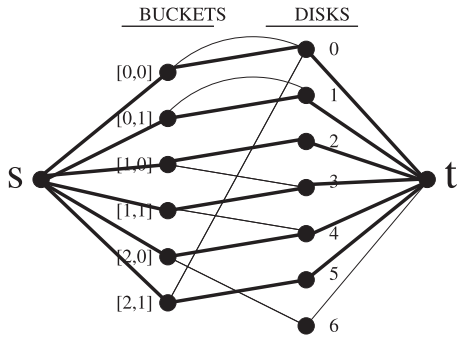


Fig. 4. Max-flow representation of query q_1 for single site.

buckets and $\lceil \frac{6}{7} \rceil = 1$, all the edges have capacity 1 in this case. When using the maximum flow representation, if the maximum flow between the source and the sink is $|Q|$, then the query can be retrieved using $\lceil \frac{|Q|}{N} \rceil$ accesses. Otherwise we need to increment the capacities of all the edges between the disks and the sink by 1 and rerun the max-flow algorithm until the flow of $|Q|$ is reached. The max-flow algorithm is called $O(|Q|)$ times in the worst case, where all the buckets are stored in a single disk.

THEOREM 4.1. *There exists a flow of size $|Q|$ from the source to the sink in the flow graph if and only if there exists a solution to the bucket retrieval problem.*

PROOF. Given in Chen and Rotem [1994] for Theorem 3.1. □

5. GENERALIZED RETRIEVAL PROBLEM

In order to have a general solution for the problem of finding the optimal response time retrieval of a given query, we need to handle the following issues.

— *Heterogeneous Disks.* Different types of disks, might have different response times depending on their rotational speeds (7200, 10,000, 15,000 RPM etc.), interface (SCSI, IDE etc.), underlying technology (HDD, SSD etc.) and so on. Retrieval from the fastest disk possible in a situation, in which all the other factors are the same is preferable.
— *Multisite Retrieval and Network Delay.* Data to be retrieved may be distributed among multiple servers and location-based server selection can be used to improve the retrieval performance. The solution must ensure that if all the other properties are the same, the disk j with the lowest network delay d_j is chosen for retrieval.
— *Initial Load.* A disk might already have an initial load to be retrieved from previous queries. If all the other factors are the same for a set of disks, the disk with the minimum or no initial load should be chosen for retrieval.

Our aim is coming up with the optimal response time retrieval of a given query for a situation involving any combination of the issues discussed in the preceding. We propose two fundamentally different approaches to the generalized retrieval problem. Our first approach uses the LP model and the second approach uses the maximum flow formulation. The main reason behind starting with the LP approach was to identify whether the problem is solvable. We first attacked the problem with the LP model since the parameters of the system can be easily formulated in LP. Being able to formulate the problem with LP and obtaining the optimal response time gave us confidence to attack the problem with a more efficient maximum flow approach. Besides, solving the same problem using two different approaches will allow us to test the correctness of the

approaches. Since both LP and max-flow techniques guarantee the optimal retrieval schedule, response times returned by them for a given query should match.

5.1. LP Solution

In this section, we will explain how to formulate the generalized retrieval problem in LP. Note that the initial load X_j and the network delay D_j for a disk j should be considered in the cost function of disk j only if disk j is used for retrieval. We can ensure this by using binary indicator variables. Therefore, we define a binary indicator variable I_j for each disk j, which takes 0 when the disk load L_j is zero, 1 when $L_j > 0$. C_j denotes the average time to retrieve a single bucket from the disk j.

As in the formulation of the basic problem, we have N different cost functions r_j for each disk j that measure the retrieval cost of a given query.

$$r_j = I_j * (D_j + X_j) + Cj * L_j, \quad j = 1, \ldots, N. \tag{5}$$

We want to

$$\begin{aligned} &Minimize : Maximum\{r_1, r_2, \ldots, r_N\} \\ &Subject\ to : r_j = I_j * (D_j + X_j) + Cj * L_j,\ j = 1, \ldots, N \\ &\qquad \sum_{j=1}^{N} B_{ij} = 1, \quad i = 1, \ldots, |Q|. \end{aligned} \tag{6}$$

Similar to (2), the problem as defined here is not a linear program but can be made into one by defining a new variable $R \geq r_j, j = 1, \ldots, N$, and minimizing R. We may rewrite this as

$$R - I_j * (D_j + X_j) - Cj * L_j \geq 0, \quad j = 1, \ldots, N. \tag{7}$$

Thus, we can reformulate (6) as the linear program

$$\begin{aligned} &Minimize : R \\ &Subject\ to : R - I_j * (D_j + X_j) - Cj * L_j \geq 0, \quad j = 1, \ldots, N \\ &\qquad \sum_{j=1}^{N} B_{ij} = 1, \quad i = 1, \ldots, |Q|. \end{aligned} \tag{8}$$

For the formulation of the generalized case, beside the $c * |Q| + 1$ variables used in the formulation of the basic case, we need N additional indicator variables making a total of $c * |Q| + N + 1$ unique variables. D_js, X_js, and C_js are constants, r_js and L_js are redefinitions of current variables. In terms of constraints, the generalized case requires N extra constraints for the indicator constraints. Therefore, we use a total of $2N + |Q|$ constraints in the formulation. In CPLEX [CPLEX], indicator constraints can be specified as follows.

$$I_j = 0 \rightarrow L_j = 0 \tag{9}$$

5.2. Max-Flow Solution

In this section, we will explain how to solve the generalized retrieval problem using maximum flow techniques. For a given flow graph G, let E be the edge set holding every edge e_j between the disk vertex j and the sink, and $caps(e_j)$ be the capacity of the edge e_j; for $j = 1, \ldots, N$. In the general case, time to retrieve b buckets from disk j can be calculated using the following cost function:

$$cost(e_j, b) = D_j + X_j + b * C_j, \tag{10}$$

where D_j denotes the network delay to the site where disk j is located, X_j denotes the initial load of disk j, and C_j denotes the average time to retrieve a bucket from disk j. In order to find the optimal response time of a given query, we should find $caps(e_j)$ for $j = 1, \ldots, N$, where G has a max-flow of $|Q|$ and the max of $cost(e_j, caps(e_j))$ is minimized.

In the basic problem, since all the disks were homogeneous without having any network delay or initial load, their cost functions to retrieve the same number of buckets were equal to each other. Therefore, we could initially set the capacities of all the edges in E to the theoretical lower bound $\lceil \frac{|Q|}{N} \rceil$, and if the flow of $|Q|$ cannot be reached, we could increment these capacities all at the same time. However, the cost functions of the disks may be different for the generalized problem since each disk may have different retrieval performance, different initial load, and different network delay. Therefore, capacities of the edges in E should be set carefully depending on the cost function of each disk in the system.

First, we will explain the construction of the flow graph for the general retrieval problem. Next, capacity incrementation algorithm of the edges in E considering the cost functions of the disks will be presented. And finally, in order to speed up the capacity incrementation process, a binary capacity scaling algorithm that sets the capacity of the edges in E using a binary search alike technique will be provided.

5.2.1. Flow Graph Construction. For retrieval of a query Q in the general case, maximum flow representation needs $|Q| + N + 2$ vertices and $|Q| * (S + 1) + N$ edges, where S is the number of sites and N is the total number of disks in the system. Assuming N_i represents the number of disks at site i; the total number of disks can be calculated as follows: $N = \sum_{i=1}^{S} N_i$.

ALGORITHM 1: *ConstructFlowGraph*()

Inputs: Q, N
Outputs: $G, E, caps, s, t$

```
1:  G = createEmptyGraph()
2:  for i ← 0 to N + |Q| + 1 do
3:     v[i] = G.new_vertex()
4:  s = v[0]
5:  t = v[N + |Q| + 1]
6:  for i ← 1 to |Q| do
7:     e = G.new_edge(s, v[i])
8:     caps[e] = 1
9:     for j ← |Q| + 1 to |Q| + N do
10:       if bucketAtDisk(i,j) then
11:          e = G.new_edge(v[i], v[j])
12:          caps[e] = 1
13: for i ← |Q| + 1 to |Q| + N do
14:    e = G.new_edge(v[i], t)
15:    caps[e] = 0
16:    E.insert(e)
```

A flow graph is constructed using Algorithm 1. By taking $|Q|$ and N as inputs, Algorithm 1 creates the flow graph G with source s and the sink t, a capacity array *caps*, and an edge set E. *caps* holds the capacities of the edges in G, and E holds all the edges between the disk vertices and the sink. Lines 1–5 create an empty graph, add all required vertices to the graph and specify the source vertex s and the sink vertex t. Then lines 6–8 add an edge from s to every vertex representing the query buckets and

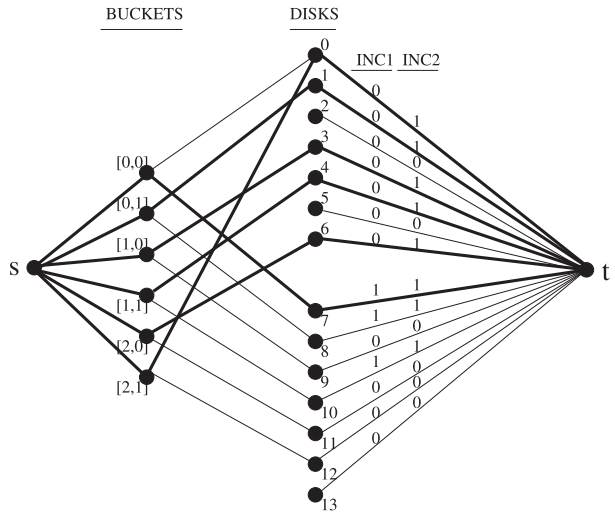


Fig. 5. Max-flow representation of query q_1 for two sites.

set the capacities of these edges to 1. Lines 9–12 add an edge from every vertex representing the query buckets to the vertices representing the disks if the bucket is stored at that disk and also set the capacities of these edges to 1. Finally, lines 13–15 add an edge to the graph from every vertex representing the disks to t with 0 capacities. These edges are also added to the edge set E at line 16.

Consider the query q_1 given in Figure 3 again but this time assume that the grid on the left represents the allocation at site 1 and the grid on the right represents the allocation at site 2. Max-flow representation of the query $q1$ is given in Figure 5. There are 14 disks in the system, disks 0–6 are located at site 1 and disks 7–13 are located at site 2.

5.2.2. Capacity Incrementation. Since the cost functions of the disks may vary in the general case, capacities of the edges in E cannot be incremented all at the same time. Cost functions of the disks should be considered in this capacity incrementation process. Starting from 0 capacities for the edges in E, we use the following two steps repeatedly to reach the final capacity values yielding the optimal response time retrieval in the general case.

(1) Increment the capacity of the edge in E yielding the minimum cost, which can be computed as $Minimum\{cost(e_j, caps(e_j) + 1)\}$ for $j = 1, \ldots, N$ using the cost functions given in (10). If more than one edge yields the minimum cost, increment them together.
(2) Run the max-flow algorithm for the capacities calculated in step 1. Stop if max-flow is $|Q|$, go to step 1 otherwise.

The capacity incrementation algorithm explained using these steps above is given in Algorithm 2. We made the following two observations in order to speed up the algorithm.

OBSERVATION 1. *There is no need to run the max-flow algorithm if the total of the capacities of the edges in E is less than* $|Q|$.

PROOF. Since E holds all the edges going to the sink, maximum flow of $|Q|$ cannot be reached if the total of the capacities of the edges going to the sink is less than $|Q|$. □

ALGORITHM 2: *CapacityIncFlow*()

Inputs: Q, G, E, D, X, C, s, t
Output: $caps, flows$

```
1: for all e ∈ E do
2:   total_caps += caps[e]
3: repeat
4:   min_cost ← MAXDOUBLE
5:   for all e ∈ E do
6:     v ← G.source(e)
7:     if G.in_degree(v) ≤ caps[e] then
8:       E.delete(e)
9:     else
10:      cost[e] ← D[e] + X[e] + (caps[e] + 1) ∗ C[e]
11:      if costs[e] < min_cost then
12:        min_cost ← costs[e]
13:  for all e ∈ E do
14:    if costs[e] == min_cost then
15:      caps[e]++
16:      total_caps++
17:  if total_caps ≥ |Q| then
18:    flow_value ← MaxFlow(G, s, t, caps, flows)
19: until flow_value == |Q|
```

Table II. System Parameters

Disk j	C_j (ms)	D_j (ms)	X_j (ms)
0–6	8.3	2	1
7,8,10,13	6.1	1	0
9,11,12	13.2	1	0

OBSERVATION 2. *Let v_j denote the source vertex of the edge $e_j \in E$ and $in\_degree(v_j)$ denote the incoming degree of the vertex v_j. There is no need to increment $caps(e_j)$ if $in\_degree(v_j) \leq caps(e_j)$.*

PROOF. Since $e_j \in E$, v_j represents the disk j and $in\_degree(v_j)$ represents the total number of buckets stored in disk j for a given query $|Q|$. The maximum amount of buckets we can retrieve from a disk is limited by the number of query buckets stored in that disk. □

Observation 1 is checked in line 17 and Observation 2 is checked in lines 6–8 of Algorithm 2. Note that in line 8, we do not remove the edge from the graph but remove it from our edge set E. Lines 10–12 determine the edges in E yielding the minimum retrieval cost and lines 14–15 increment the capacities of the edges with this minimum cost. Note that, if more than one edge yields the same retrieval cost, their capacities are incremented at the same time as in the basic problem.

For query q_1 given in Figure 3 and the set of parameters summarized in Table II, the flow graph giving the optimal response time retrieval calculated using Algorithm 2 is given in Figure 5. E holds all the edges between the disk vertices and the sink vertex t. All the capacities of the edges in E are 0 at the beginning. First, the edge between the disk vertex 13 and the t, e_{13}, is deleted from E as in line 7–8 of the algorithm based on the Observation 1. Then, costs for each edge in E are calculated as in line 10 of the algorithm. Since $min\_cost = 7.1$ for disks 7, 8, and 10 the capacities of e_7, e_8, and e_{10} are incremented by 1, as shown by INC1 in the figure. For these capacities, there is no need to run the max-flow algorithm based on Observation 2 since $total\_caps = 3$ is

smaller than $|Q| = 6$. For the second incrementation step, $min\_cost = 11.3$ for disks 0–6; the capacities of e_0-e_6 are incremented by 1 as shown by INC2 in the figure and the max-flow is run this time. Max-flow returns 6 and the algorithm stops, since the $flow\_value = 6 = |Q|$. This flow is shown with thick lines in the figure. Optimum response time of q_1 is 11.3 ms by retrieving the buckets from the disks that are connected with thick lines in the figure. By using the final flow graph, the optimal response time is calculated as $R = Maximum\{cost(e_j, flow(e_j))\}$ for $j = 1, \ldots, N$ using the cost function given in (10).

THEOREM 5.1. *Starting with* 0 *capacities for every edge between a disk vertex and the sink, the response time R calculated using the flow graph returned by Algorithm 2 is optimal.*

PROOF. Algorithm 2 starts with 0 capacities. In each incrementation step, it only increments the capacity of the edge(s) yielding the next minimum retrieval time. After each incrementation step, it calls max-flow to check the maximum flow of $|Q|$. The algorithm stops for the first capacity set yielding the maximum flow of $|Q|$. Since it considers all possible retrieval times starting from the minimum in an exhaustive search manner and stops when the maximum flow of $|Q|$ is reached, R calculated using the flow graph returned by Algorithm 2 is optimal. □

Although Algorithm 2 minimizes the amount of max-flow calls and the capacity incrementation using the observations, it will perform $O(c|Q|)$ max-flow calls in the worst case. The number of max-flow calls is directly dependent on the number of incrementation steps carried out. Since we start from 0 capacities, we may end up with the worst case number of max-flow calls if the cost functions of the disks are unique. In order to improve the performance of Algorithm 2, we propose the binary capacity scaling algorithm.

5.2.3. Binary Capacity Scaling. The challenge of the generalized retrieval problem is coming up with the capacity values of the edges in E that yield the optimal response time R. If we know these capacities, the problem is easy since calling the max-flow algorithm returns the optimal retrieval schedule. Algorithm 2 finds the capacity values that yield R in an exhaustive search manner; however we need a more efficient algorithm since obtaining the optimal retrieval schedule is a time critical issue. In order to speed up Algorithm 2, we propose the binary capacity scaling method. The main idea behind binary capacity scaling is bringing the capacity values up to an initial value before the incrementation step is started by Algorithm 2. In this way, we will have fewer incrementation steps, and consequently fewer max-flow calls. Observation 3 constitutes the base of the binary capacity scaling technique.

OBSERVATION 3. *Given a flow graph G. For every time value t, there exists a corresponding capacity set holding the capacity of the edges in E. For a time value t, this capacity set is calculated as* $caps(e_j, t) = \lfloor \frac{t - D_j - X_j}{C_j} \rfloor$ *for every edge* e_j *between the disk j and the sink.*

Observation 3 is described using Figure 6 for one site assuming arbitrary values of C_j, D_j, and X_j for disk number j given on the x-axis. Distribution of the buckets on the disks is given on the left and the corresponding capacities of the flow graph are given on the right. For a given time t, calculated capacities using Observation 3 represent the maximum number of buckets that can be retrieved from a disk during the time t. The network delay shown with the horizontal line is the same for all the disks since they are all located on a single site; however, initial loads of the disks may vary as shown by the horizontal patterned black blocks located on some of the disks. The dashed empty

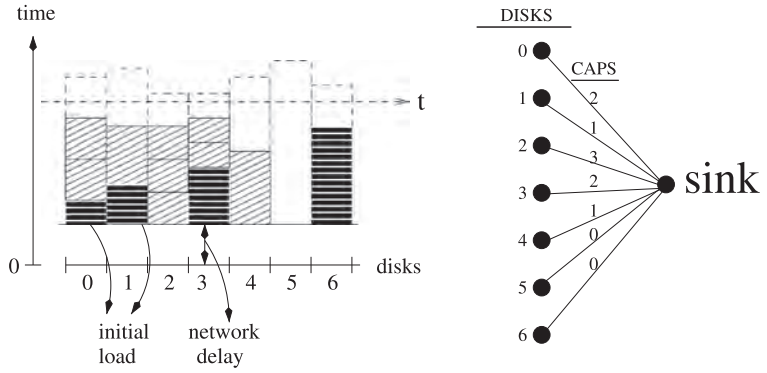


Fig. 6. Capacity distribution for a time t.

blocks for each disk represent the time required for the retrieval of the next bucket that cannot be retrieved in the given time t. For instance, disk 5 is the slowest disk in the system since retrieval of even one bucket requires more than t amount of time.

Based on Observation 3, we can state the following lemmas.

LEMMA 5.2. *Given a flow graph G, the optimal response time R is the minimum time t that G has a maximum flow of $|Q|$ using the capacities $caps(e_j, t)$ for every edge e_j between the disk j and the sink.*

PROOF. From the definition of optimal response time. □

LEMMA 5.3. *Given a flow graph G and time values t_1, t_2, if $t_1 \leq t_2$, then $caps(e_j, t_1) \leq caps(e_j, t_2)$ for every edge e_j between the disk j and the sink.*

PROOF. Follows from Observation 3. □

By using Observation 3, we can calculate the corresponding capacities of the flow graph for a given time value t. If these capacities yield a maximum flow of less than $|Q|$, then we can conclude that some of the query buckets cannot be retrieved in this given time t. In order to retrieve these nonretrieved buckets, we can use Algorithm 2 for capacity incrementation starting from the capacities calculated for t. In other words, Theorem 5.1 can be generalized as follows.

THEOREM 5.4. *Given a flow graph G and a time value t, assume that the maximum flow of G is less than $|Q|$ for the capacity values $caps(e_j, t)$ for every edge e_j between the disk j and the sink. Then, starting with these capacities, the response time R calculated using the flow graph returned by the Algorithm 2 is optimal.*

PROOF. Follows from Theorem 5.1. We know that the capacity set found for a t value using Observation 3 represents the maximum number of buckets that can be retrieved from each disk during the time t. In other words, capacities calculated represent one of the minimum retrieval time values that Theorem 5.1 searches through. Starting with 0 retrieval time, the minimum retrieval time values calculated after each incrementation step in Theorem 5.1 can be considered as a linear chain of values leading to the optimal response time R. Each value in the chain is followed by only one unique value without causing forks in the chain. Since the chain is linear, starting from any point of the chain and using Algorithm 2 will lead us to the optimal response time R. □

By using Theorem 5.4, we can reach R with the initial capacity values calculated for a time value t. However, our challenge here is to find the time value t that is close

enough, but less than, R in an efficient way so that the number of incrementation steps performed by Algorithm 2 is minimized. For this problem, we are using the binary range reduction technique given in Theorem 5.5.

THEOREM 5.5. *Given a flow graph G for a query Q. Using Observation 3; define an upper range t_{max} where G has a maximum flow of $|Q|$, and define a lower range t_{min} where G has a maximum flow of less than $|Q|$ such that the optimal response time R is within the range, $t_{min} < R \leq t_{max}$. If the range is reduced by half using the middle value $t_{mid} = t_{min} + (t_{max} - t_{min}) * 0.5$, then R is also within this reduced range, $t_{min} < R \leq t_{max}$; where if G has a maximum flow of $|Q|$ using the capacities calculated for t_{mid}, then $t_{max} = t_{mid}$, else $t_{min} = t_{mid}$.*

PROOF. Upper range is $[t_{mid}, t_{max}]$ and lower range is $[t_{min}, t_{mid}]$. We know that maximum flow is $|Q|$ for t_{max} and less than $|Q|$ for t_{min}. If the maximum flow is $|Q|$ for t_{mid}, then we can skip all the time values in the upper range since we are looking for the minimum time value satisfying the maximum flow of $|Q|$ by Lemma 5.2. We can set t_{max} to t_{mid} knowing that the maximum flow is still $|Q|$ for the new t_{max}. If the maximum flow is less than $|Q|$ for t_{mid}, then we can skip all the time values in the lower range since they will yield smaller or equal capacity values by Lemma 5.3 and we cannot reach the maximum flow of $|Q|$ using these capacity values. In this case, we can set t_{min} to t_{mid} knowing that the maximum flow is still less than $|Q|$ for the new t_{min}. □

The range reduction process described in Theorem 5.5 can be applied repeatedly to reduce the range further and t_{min} can be used to calculate the initial capacities at the end. However, a stopping case is required. We stop the range reduction process when the range is less than or equal to the time of retrieving a bucket from the fastest disk in the system, $Min\{C_j\}$ for $j = 1 \ldots N$. In this way, we make sure that the final incrementation steps performed by Algorithm 2 will be bounded by the number of disks in the worst case.

The binary capacity scaling algorithm is given in Algorithm 3. The algorithm first defines the range $[t_{min}, t_{max}]$ that the optimal retrieval time lies within, in lines 5–8. For t_{max}, all the blocks are assumed to be retrieved from a single disk with the maximum cost; and for t_{min}, $\lceil \frac{|Q|}{N} \rceil$ of the blocks are assumed to be retrieved from a single disk with the minimum cost. Since the maximum flow for t_{min} should be less than $|Q|$, we subtract the *min_speed* value from t_{min} in line 11. *min_speed* is the average retrieval time of a single block from the fastest disk in the system calculated in lines 9–10.

The algorithm finds the capacities of the flow graph for t_{mid} in line 16 and calls the max-flow algorithm in line 17. If there is a solution such that $flow\_value == |Q|$, then t_{max} is decreased to t_{mid} in line 19; otherwise, t_{min} is increased to t_{mid} in line 22. The algorithm stops when the range is smaller than *min_speed* and calculates the final capacities using the t_{min} of the final range in line 23–24.

Investigation on the position of the optimal response time within the initial $[t_{min}, t_{max}]$ range led us to use the r value, which is used in line 14 for the calculation of t_{mid}, closer to t_{min} until t_{max} is decreased for the first time. In this way, we can decrease the number of max-flow calls. As a result of the experimentation on the position of the optimal response time within the initial range, we found out that $r = \frac{2.5}{N}$ as in line 12 of the algorithm is a good approximation for the general case.

Consider query q_2 given in Figure 3 for the system parameters described in Table II. In Figure 7, optimal response time retrieval of the query q_2 is shown. The blocks are located on the disks that they are retrieved from and the block heights vary depending

ALGORITHM 3: *BinaryCapScale*()

Inputs: Q, G, E, D, X, C, s, t
Output: $caps, flows$

1: $min\_speed \leftarrow MAXDOUBLE$
2: $t_{min} \leftarrow MAXDOUBLE$
3: $t_{max} \leftarrow 0$
4: **for all** $e \in E$ **do**
5: **if** $D[e] + X[e] + |Q| * C[e] > t_{max}$ **then**
6: $t_{max} \leftarrow D[e] + X[e] + |Q| * C[e]$
7: **if** $D[e] + X[e] + \lceil \frac{|Q|}{N} \rceil * C[e] < t_{min}$ **then**
8: $t_{min} \leftarrow D[e] + X[e] + \lceil \frac{|Q|}{N} \rceil * C[e]$
9: **if** $C[e] < min\_speed$ **then**
10: $min\_speed \leftarrow C[e]$
11: t_{min} -= min_speed
12: $r \leftarrow \frac{2.5}{N}$
13: **while** $(t_{max} - t_{min}) \geq min\_speed$ **do**
14: $t_{mid} \leftarrow t_{min} + (t_{max} - t_{min}) * r$
15: **for all** $e \in E$ **do**
16: $caps[e] \leftarrow \lfloor (t_{mid} - D[e] - X[e]) / C[e] \rfloor$
17: $flow\_value \leftarrow MaxFlow(G, s, t, caps, flows)$
18: **if** $flow\_value == |Q|$ **then**
19: $t_{max} \leftarrow t_{mid}$
20: $r \leftarrow 0.5$
21: **else**
22: $t_{min} \leftarrow t_{mid}$
23: **for all** $e \in E$ **do**
24: $caps[e] \leftarrow \lfloor (t_{min} - D[e] - X[e]) / C[e] \rfloor$

on the retrieval performance of the disk. Optimal response time is found to be 19.6 ms using the retrieval schedule given in the figure. The initial range calculated by Algorithm 3 for this specific example is shown in the figure using $max = 370.6$ and $min = 7.1$. The max value is calculated assuming that all the blocks are only allocated in the disk with the maximum cost; however, this is a very low probability. Using $r = \frac{2.5}{14} = 0.17$ for the calculation of the first middle value yields $mid1 = 72$. Since the graph using the capacities calculated for the time value $mid1 = 72$ has a maximum flow of $|Q|$, max is set to $mid1$ for the next range. After decreasing max to $mid1$, we set r to 0.5 as in line 20 of the algorithm since the optimal response time can now be anywhere in the range. However, since the graph does not have a maximum flow of $|Q|$ for $mid5 = 19.2$, min is set to $mid5$ for the final range. The algorithm stops for the range $[mid5, mid3]$ since the range $(mid5 - mid3 = 4.1)$ is smaller than the $min\_speed = 6.1$, which denotes the time of retrieving a bucket from the fastest disk in the system. By using the initial capacities calculated for $mid5$, Algorithm 2 will only need 2 incrementation steps to reach the optimal response time of 19.6. The first incrementation step is for disks 7, 10, 13; and the second incrementation step is for disks 0–6. Since the costs of the disks are equal to each other in each incrementation step, the capacities are incremented together.

The proposed max-flow solution is given in Algorithm 4. First the flow graph is constructed using Algorithm 1 in line 2. The next step is scaling the capacities using Algorithm 3 in line 3. Then, the scaled capacity values stored in *caps* are incremented until a maximum flow of $|Q|$ is reached using Algorithm 2 in line 4. Optimal response time is calculated using the flow graph returned by Algorithm 2 through lines 5–9.

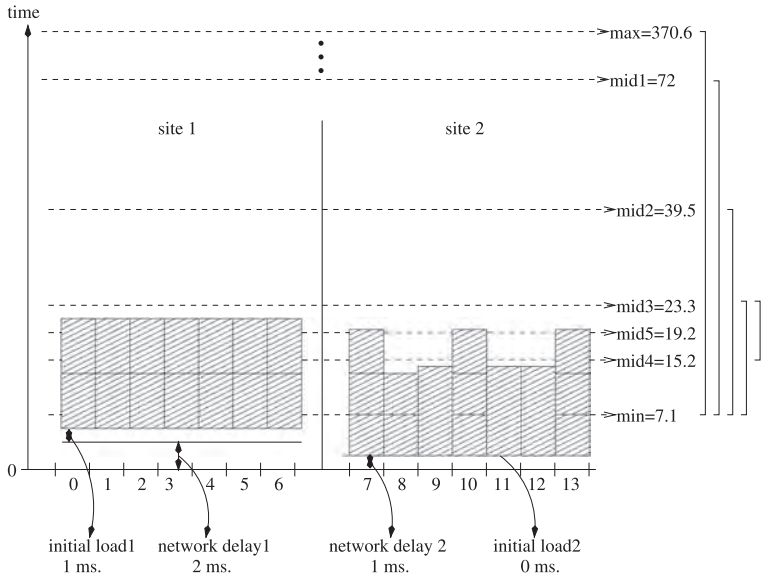


Fig. 7. Binary Capacity Scaling for query q_2.

ALGORITHM 4: *MaxflowGeneralizedRetrieval*()

Inputs: Q, N, D, X, C
Output: R

```
1: R ← 0, flows ← 0, caps ← 0
2: ConstructFlowGraph()
3: BinaryCapScale()
4: CapacityIncFlow()
5: for all e ∈ G do
6:    if flows[e] > 0 then
7:       edge_cost ← D[e] + X[e] + flows[e] * C[e]
8:       if edge_cost > R then
9:          R ← edge_cost
```

COROLLARY 5.6. *The R value returned by Algorithm 4 is the optimal response time for the generalized retrieval problem. Assuming B_{ij} is 1 if bucket i is retrieved from disk j and 0 otherwise, the optimal retrieval can be obtained by simply setting B_{ij} to the flow value between the vertex v_i denoting bucket i and the vertex v_j denoting disks j.*

Corollary 5.6 follows from Theorems 5.4 and 5.5.

5.3. Online Algorithm

In this section, we propose a low complexity online algorithm for the generalized retrieval problem given in Algorithm 5. As distinct from the LP and the max-flow solutions, the online algorithm does not guarantee the optimality of the result. On the other hand, it is simpler and has a lower time complexity than the other two solutions. The algorithm makes a final retrieval decision for a bucket in a round-robin fashion by considering retrieval from the site introducing the minimal cost. The function *DiskAtSite*(i,j) returns the disk vertex k bucket, i is stored in site j, and $load[k]$ holds the number of buckets scheduled for disk k. The site that leads to the earliest retrieval time is chosen for retrieval of a bucket.

ALGORITHM 5: *OnlineGeneralizedRetrieval*()

Inputs: Q, N, S, D, X, C
Output: R

```
1: R ← 0
2: for i ← 1 to N do
3:    load[i] ← 0
4: for i ← 1 to |Q| do
5:    min_cost ← MAXDOUBLE
6:    for j ← 1 to S do
7:       k = DiskAtSite(i, j)
8:       cost = D[k] + X[k] + (load[k] + 1) * C[k]
9:       if cost < min_cost then
10:         min_cost ← cost
11:         min_disk_index ← k
12:   load[min_disk_index]++
13:   if min_cost > R then
14:      R ← min_cost
```

5.4. Complexity Analysis of the Algorithms

The LP solution provided in this article is a special case of linear programming called binary integer programming, classified as NP-hard [Karp 1972]. Complexity of the max-flow solution described in Algorithm 4 depends on the complexity of the maximum flow algorithm used, which can be solved in polynomial time. The number of max-flow calls performed by Algorithm 4 in the worst case is $O(log(|Q|) + N)$, where $O(log(|Q|))$ comes from the binary capacity scaling part and $O(N)$ comes from the capacity incrementation part. The online algorithm's worst case complexity is $O(S|Q|)$ without requiring any max-flow computation.

6. EXPERIMENTAL RESULTS

In this section, we provide the experimental results using various sets of parameters. We investigate the impact of allocation scheme, query type, query load, disk speed, network delay to the site, and initial load of the disks, on the experimental results. In all experiments with synthetic workload, we used an $N \times N$ grid and N disks per site. Experiments with the real-world workloads are performed using a 100×100 grid and the actual number of disks specified in the workload per site. We implemented the algorithms in C++ and compiled using g++ optimization level 1 (-O1). For the graph structure and the max-flow computation, we used the LEDA library version 3.4.1 [Mehlhorn and Näher 1995], which implements the max-flow algorithm using the push-relabel method proposed by Goldberg and Tarjan [1988] with time complexity of $O(V^3)$ for V vertices. For the LP model, we used the CPLEX Academics 12.3 [CPLEX] MIP solver, which uses the branch and bound method.

6.1. Allocation Scheme

We describe the three different allocation schemes used in our experiments in the following.

— *Random Duplicate Allocation.* Random Duplicate Allocation (RDA) [Sanders et al. 2000] stores a bucket on two disks chosen randomly from the set of disks. The retrieval cost of random allocation is at most 1 more than the optimal, with high probability for single site retrieval.
— *Orthogonal Allocation.* Orthogonal allocations [Ferhatosmanoglu et al. 2004; Tosun 2004] guarantee that when the disks that a bucket is stored at are considered as a

pair, each pair appears only once in the disk allocation. In an $N \times N$ declustering system with N disks, there are N^2 buckets and N^2 pairs. So it is possible to have each pair exactly once. For the first copy, we used the threshold-based declustering scheme [Tosun 2007b].

— *Dependent Periodic Allocation.* A d-dimensional disk allocation scheme $f(i_1, i_2, \ldots, i_d)$ is *periodic* if $f(i_1, i_2, \ldots, i_d) = (a_1 * i_1 + a_2 * i_2 + \ldots + a_d * i_d) \ mod \ N$, where N is the number of disks and each a_i $i = 1 \ldots d$ satisfies $gcd(a_i, N) = 1$ and $a_i \neq 0$ [Altiparmak and Tosun 2012; Tosun and Ferhatosmanoglu 2002]. For the first copy, we use the allocation scheme yielding the lowest additive error based on the results provided in Altiparmak and Tosun [2012]. For the second copy, a shifted version of the first copy is used. The two allocations are of the form $f(i,j) = a_1 * i + a_2 * j \bmod N$ and $g(i,j) = f(i,j) + m \bmod N$, $1 \leq m \leq N-1$.

6.2. Query Types

We now describe the three different query types we used in our experiments.

— *Range Query.* Range queries are rectangular in shape. We assume a wraparound grid consistent with the choice of disk allocations. A range query is identified with 4 parameters (i,j,r,c) $0 \leq i,j \leq N-1, 1 \leq r,c \leq N$. i and j are indices of the top left corner of the query and (r, c) denote the number of rows and columns in the query. The number of distinct range queries on an $N \times N$ grid is $(\frac{N*(N+1)}{2})^2$, which can be found by counting the number of ways to choose two points out of $N+1$ row and column points on the grid as follows: $\binom{N+1}{2} * \binom{N+1}{2}$.

— *Arbitrary Query.* Arbitrary queries have no geometric shape. Any subset of the set of buckets is an arbitrary query. We can denote arbitrary queries as a set and the number of arbitrary queries is $\sum_{i=1}^{N^2} \binom{N^2}{i}$ which is equal to 2^{N^2} (number of subsets of a set with N^2 elements).

— *Connected Query.* The buckets in a connected query form a connected graph. Create a node for each bucket in the query and connect two buckets $[i,j]$ and $[m,n]$ by an edge if they are neighbors in the wraparound grid. If the resulting graph is connected, then it is a connected query.

6.3. Query Load

We use three synthetic, and one real-world, query loads. For the synthetic workloads, we use the notation p_k^i to denote the probability that a query in load i can be retrieved in k disk accesses optimally. Once the optimal number of disk accesses k is selected, the number of buckets is selected uniformly from the range $[(k-1)N + 1, kN]$. As a real world workload, we use the popular Exchange workload representing Microsoft's production mail server. Query size is determined using the real request sizes of the trace.

— *Load 1.* The distribution of queries is similar to the distribution of queries for the particular query type. For the distribution of range queries, smaller size queries are more likely; for the distribution of arbitrary queries, medium size queries are more likely. We use the distribution of range queries for connected queries, since it is hard to find the distribution of connected queries. The expected bucket size of load 1 queries is $\frac{N^2}{4} + O(\frac{1}{N})$ for range queries and $\frac{N^2}{2} + O(\frac{1}{N})$ for arbitrary queries.

— *Load 2.* The distribution of queries is uniform. We achieve this by setting $p_k^2 = \frac{1}{N}$. The expected bucket size of load 2 queries is $\frac{N^2}{2}$.

Table III. Disk Specifications

Producer	Model	Type	RPM	Seek T.	Latency	Bandwidth	Avg. Access Time
Seagate	Barracuda	HDD	7.2 K	8.5 ms	4.1 ms	57 MB/s	13.2 ms
WD	Raptor	HDD	10 K	4.2 ms	5.5 ms	68 MB/s	8.3 ms
Seagate	Cheetah	HDD	15 K	3.6 ms	2.0 ms	86 MB/s	6.1 ms
OCZ	Vertex	SSD	-	-	0.1 ms	197 MB/s	0.5 ms
Intel	X25-E	SSD	-	-	0.07 ms	250 MB/s	0.2 ms

—*Load 3.* Smaller queries are more likely. We achieve this by setting $p_k^3 = \frac{2^N}{(2^N-1)*2^k}$. In this case $p_k^3 = \frac{1}{2}p_{k-1}^3, 2 \leq k \leq N$. The expected bucket size of load 3 queries is $\frac{3N}{2}$.

—*Exchange.* A popular multidevice server trace previously used in various storage related studies [Agrawal et al. 2008; Narayanan et al. 2008, 2009]. It is taken from a server running the Microsoft Exchange 2007 inside Microsoft [Kavalanekar et al. 2008]. Exchange is a mail server for 5000 corporate users consisting of 9 active volumes and about 40 million block read requests. The trace covers a 24-hour weekday period starting at 2:39pm on the 12th December 2007 and is broken into 15-minute intervals. Exchange is publicly distributed via the online trace repository provided by the Storage Networking Industry Association [SNIA].

6.4. Disks

We have experimental results on five different disks. Specifications of the disks are provided in Table III. All the values except the *Average Access Time* value are obtained by the factory specifications of the disk. Average access time is the time spent to reach a bucket in a disk and calculated experimentally running Ubuntu Disk Utility's read only benchmark on the related disk.

In order to calculate the value of C_j—average retrieval cost of a single bucket from disk j—we need to consider both the average access time and the transfer time of a bucket. Transfer time of a bucket can easily be calculated using the *Bandwidth* value provided in the table. For example, the transfer time of an HDD block (512 B) from a Barracuda disk is calculated as 9 microseconds or the transfer time of an SSD block (4 KB) from a Vertex SSD is calculated as 20 microseconds. Since the average access time is the dominating factor of the retrieval cost of a bucket, it is a good metric by itself to be used as the value of C_j.

6.5. Experiment Parameters

All the experiments conducted are summarized in Table IV. R(2,10,2) means that a number among the set 2, 4, 6, 8, and 10 is chosen randomly. If the system is homogeneous, Cheetah disk is used for all the disks in the system. If the system is heterogeneous, then the disks are chosen randomly from HDDs, SSDs, or HDDs+SSDs. We choose the initial load values to be around the range [0–20] based on the average access times of the disks presented in Table III. Similarly, network delay values are chosen to be in the same range based on a study performed on a storage network presented in Orenstein [2003]. According to this study, the total estimated network latency (propagation delay + node delay + congestion delay) is found to be around 23ms. For experiment 20, all the following sites have the same properties as the first site.

6.6. Results

In this section, we provide some of the experimental results that are interesting for our purposes. All the results are available on the project web page [PW]. Experiments are performed for 1000 queries if the workload is synthetic. For real-world workloads, we

Table IV. Experiments

Exp. Num.	Num. of Sites	Disk Prop.	Site 1			Site 2		
			Disks	Delays	Loads	Disks	Delays	Loads
1	1	hom.	Cheetah	0 ms	0 ms	-	-	-
2	1	het.	SSD	0 ms	0 ms	-	-	-
3	1	het.	HDD	0 ms	0 ms	-	-	-
4	1	het.	SSD+HDD	0 ms	0 ms	-	-	-
5	1	het.	SSD+HDD	R(2,10,2)	R(2,10,2)	-	-	-
6	2	hom.	Cheetah	0 ms	0 ms	Cheetah	0 ms	0 ms
7	2	hom.	Cheetah	0 ms	0 ms	Cheetah	0 ms	20 ms
8	2	hom.	Cheetah	0 ms	5 ms	Cheetah	0 ms	15 ms
9	2	hom.	Cheetah	0 ms	10 ms	Cheetah	0 ms	10 ms
10	2	hom.	Cheetah	0 ms	15 ms	Cheetah	0 ms	5 ms
11	2	hom.	Cheetah	0 ms	20 ms	Cheetah	0 ms	0 ms
12	2	hom.	Cheetah	0 ms	0 ms	Cheetah	20 ms	0 ms
13	2	hom.	Cheetah	5 ms	0 ms	Cheetah	15 ms	0 ms
14	2	hom.	Cheetah	10 ms	0 ms	Cheetah	10 ms	0 ms
15	2	hom.	Cheetah	15 ms	0 ms	Cheetah	5 ms	0 ms
16	2	hom.	Cheetah	20 ms	0 ms	Cheetah	0 ms	0 ms
17	2	het.	SSD	0 ms	0 ms	HDD	0 ms	0 ms
18	2	het.	HDD	0 ms	0 ms	SSD	0 ms	0 ms
19	2	het.	SSD+HDD	0 ms	0 ms	SSD+HDD	0 ms	0 ms
20	2-5	het.	SSD+HDD	R(2,10,2)	R(2,10,2)	SSD+HDD	R(2,10,2)	R(2,10,2)

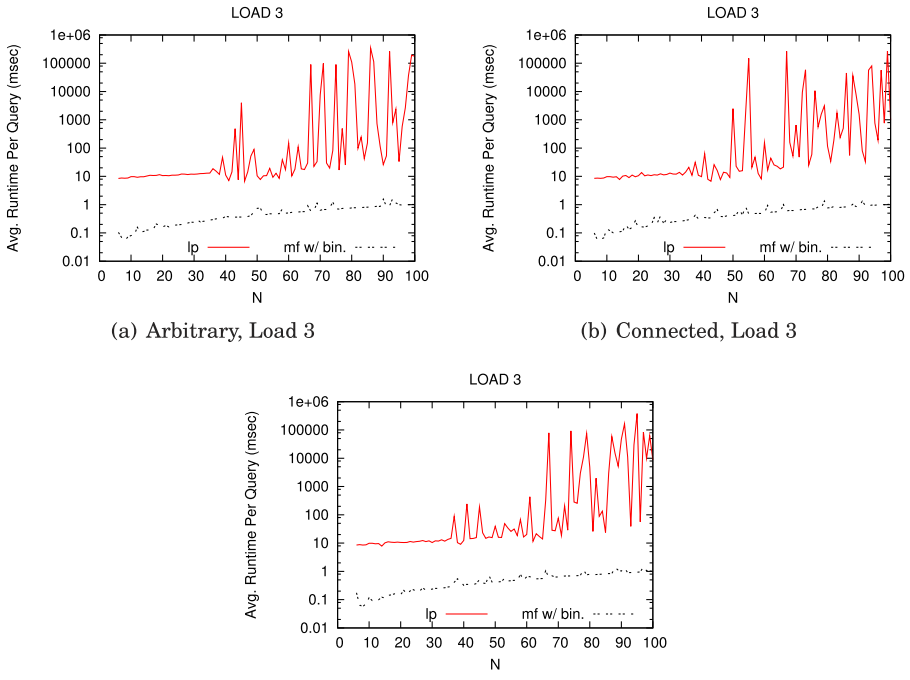
used the same number of requests performed in the workload. In most of the experiments, we used two different metrics to compare the performance of different retrieval algorithms. The first metric is the time it takes to retrieve all the queries performed in a given experiment, notated as *Retrieval Time* or *Total Retrieval Time*. Retrieval time does not include the runtime of the algorithm. Therefore, in a given experiment, *Total Retrieval Time* value is expected to be the same for the algorithms guaranteeing the optimal response time retrieval. The second metric is the the average time it takes to calculate the retrieval decision of a query, notated as *Avg. Runtime Per Query*.

We propose three fundamentally different algorithms: *LP*, *Max-flow*, and *Online*. Among the algorithms we propose, we guarantee the optimality of the *Total Retrieval Time* for *LP* and *Max-flow* (with and without the binary capacity scaling technique) approaches. We tested the *Total Retrieval Time* values returned by *LP* and *Max-flow* algorithms for Experiment 20 using two sites and found out that the results match as expected. If not stated otherwise, we only use *Max-flow* with binary capacity scaling (Algorithm 4) to calculate the *Total Retrieval Time* for the rest of the experiments, since it is has the fastest execution time.

For the *Avg. Runtime Per Query* comparisons, the machine we used has dual Intel Xeon X5672 quad-core processors with a total of 8 cores. Each core has 3.2 GHz of clock speed and the machine has 32GB of physical memory running on an Ubuntu 10.04.03 LTS operating system. The multithreading options of LEDA and CPLEX are disabled for fair comparison, therefore all the experiments use a single core only.

Initial load and network delay affect the experimental results in the same way. Therefore, the results for experiments 7, 8, 9, 10, and 11 are equivalent to the results for experiments 12, 13, 14, 15, and 16 respectively, since their ($inital\_load$ + $network\_delay$) values are equal to each other.

6.6.1. Max-Flow Solution vs. LP Solution. In this section, we compare the *Avg. Runtime Per Query* values returned by the *LP* and the *Max-flow* with the binary capacity



(a) Arbitrary, Load 3

(b) Connected, Load 3

(c) Range, Load 3

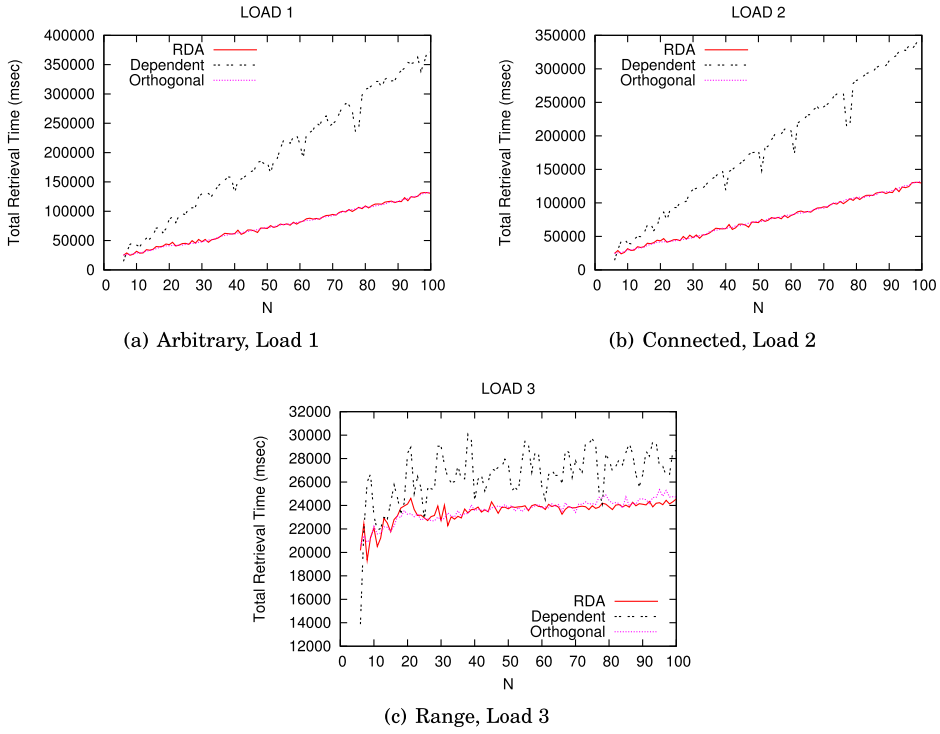
Fig. 8. Experiment 20, running time of LP vs. Max-flow, load 3, RDA, two sites.

scaling algorithm using Experiment 20 for two sites. Results are provided in Figure 8 for different query types of load 3. The x-axis shows the number of disks N per site and the y-axis shows the *Avg. Runtime Per Query*. The time value shown on the y-axis is represented on a logarithmic scale. We only present the results for the allocation scheme of RDA here since the other allocation schemes produce similar results.

Although both of the algorithms have the ability to calculate the optimal response time retrieval schedule, their execution time difference is obvious as one might also guess from their time complexities, presented in Section 5.4. It is clear from Figure 8 that *LP* is a few orders of magnitude slower than *Max-flow* depending on the number of disks and the request size.

6.6.2. Retrieval Performance of the Allocation Schemes. In this section, we investigate the retrieval performance of the different allocation schemes introduced in Section 6.1. The motivation behind providing this section is two-fold. First, proposed algorithms can calculate the optimal response time retrieval schedule without depending on the allocation scheme. Second, the allocation scheme in use for a storage array plays an important role in its performance. Since it was not possible to calculate the optimal retrieval time for the generalized case before this work, a true comparison of different allocation schemes could not be performed for heterogeneous disks having initial load and network delay. We believe that using our proposed algorithms as a metric to compare the performance of allocation schemes is another contribution of this work.

Dependent Periodic Allocation generally performs the worst for the multisite retrieval. This can be realized as a result of Experiment 20 with two sites shown in Figure 9. *Orthogonal Allocation* and *RDA* perform similar to each other but they perform generally better than the *Dependent Periodic Allocation*. The reason for this lies



(a) Arbitrary, Load 1

(b) Connected, Load 2

(c) Range, Load 3

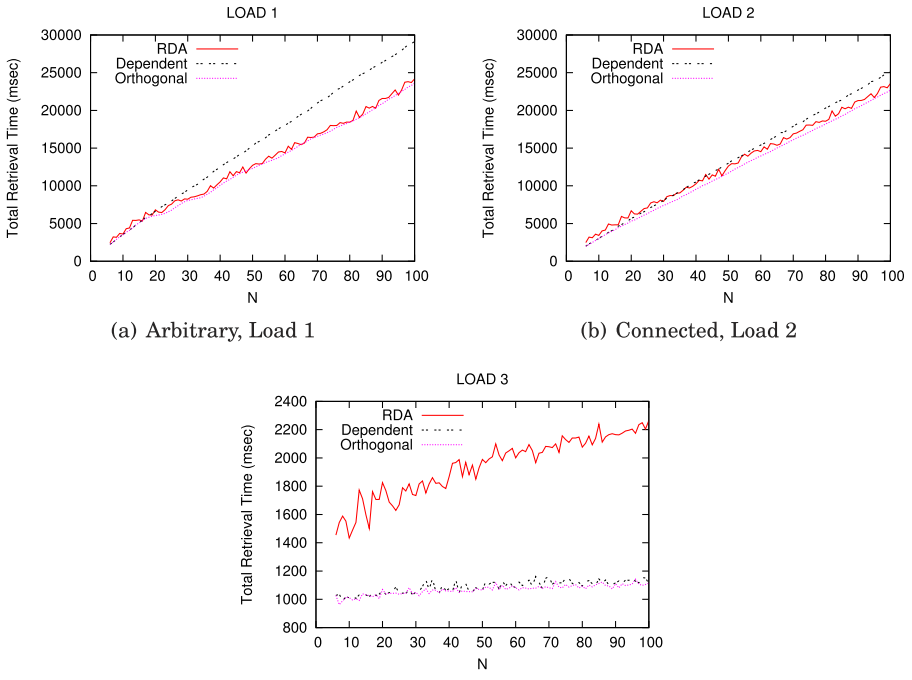
Fig. 9. Experiment 20, total retrieval time.

in the second copy chosen for the *Dependent Periodic Allocation*. Since the second copy used in site two is the shifted version of the first copy used in site one, the second site does not have the ability to complement the cases where the first site performs badly. On the other hand, *Orthogonal Allocation* seems a better scheme to be used in multi-site retrieval since the first copy used in site one is orthogonal to the second copy used in site two.

When one site is composed of SSDs and the other site is composed of HDDs, as in Experiment 17 shown in Figure 10, retrieval performance of the allocation schemes are similar to a single site retrieval as in Experiment 2 shown in Figure 11. Since the optimal response time retrieval enforces the usage of the fastest disk available in order to minimize the total retrieval time, the site with SSDs is heavily used, especially when the query size is small as in *Load 3*. For *Load 1* and *Load 2*, the allocation schemes perform similarly to each other, but the performance of *RDA* degrades significantly for small queries of *Load 3*. Allocation schemes perform in a similar way for experiments 7 (=12), 8 (=13), 10 (=15), and 11 (=16), since they also behave like a single site retrieval because of the differences between the total delay values among the two sites.

When the disks are homogeneous and the total delay values among the disks (*inital_load* + *network_delay*) are similar to each other, then the performance of the allocation schemes changes depending on the query type. This can be better observed from Experiment 9 shown in Figure 12 for small queries of *Load 3*.

— *Dependent Periodic Allocation* performs poorly for *Arbitrary* queries, but good for *Range* queries.



(a) Arbitrary, Load 1

(b) Connected, Load 2

(c) Range, Load 3

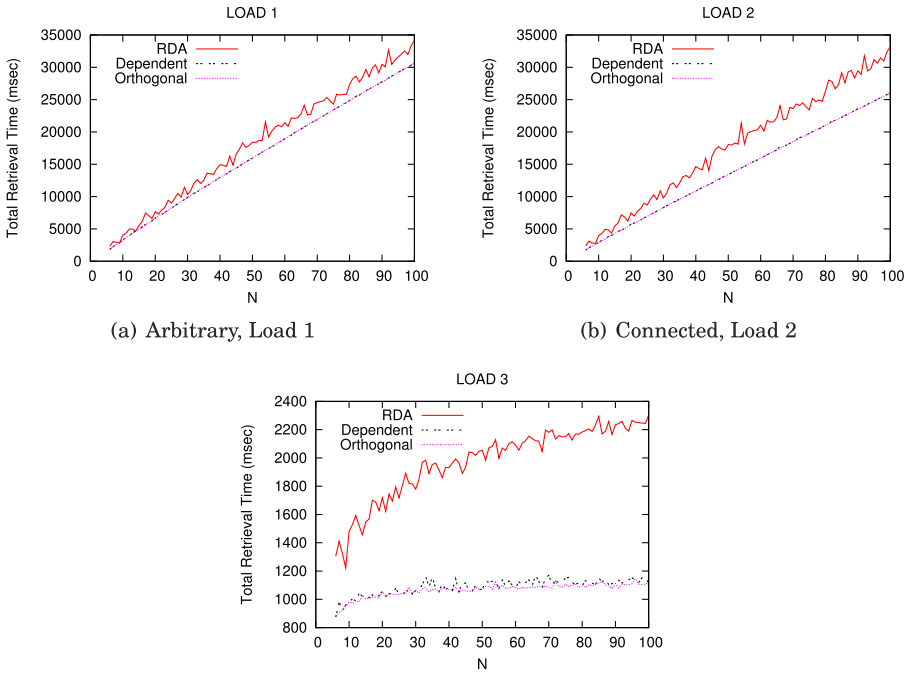
Fig. 10. Experiment 17, total retrieval time.

—*Orthogonal Allocation* performs poorly for *Connected* and *Range* queries as the number of disks increase, but good for *Arbitrary* queries.
—*RDA* performs poorly for *Range* queries, but good for *Arbitrary* and *Connected* queries.

Retrieval performance of the allocation schemes for Experiment 6 is similar to the results of Experiment 9 since the disks are also homogeneous and the total delay values of the disks for the two sites are also equal to each other in Experiment 6.

In summary, the allocation scheme to be used in a storage array should be chosen carefully depending on the characteristics of the system. First of all, *Dependent Periodic Allocation* should generally be avoided in the case of a multisite retrieval except for a few specific cases of known query types. Second, if the retrieval is known frequently to be from a single site, then *RDA* should generally be avoided if the query sizes are expected to be small. And finally, if the disks and the total delay values among the disks are more like homogeneous, then the allocation scheme should be chosen mainly depending on the expected query type of the system. In this specific case, *Dependent Periodic Allocation* can be the choice of allocation if *Range* queries are heavily expected. Otherwise, *Orthogonal Allocation* performs best if *Arbitrary* queries are more common and *RDA* performs well in many cases if *Connected* queries are more expected.

6.6.3. Online Retrieval Algorithm. The online retrieval algorithm (*Online*) does not guarantee the optimality of the retrieval time; however, how well does it perform compared to the optimal values? In this section, we compare the retrieval performance of *Online* to the optimal retrieval values calculated using *Max-flow*.



(a) Arbitrary, Load 1

(b) Connected, Load 2

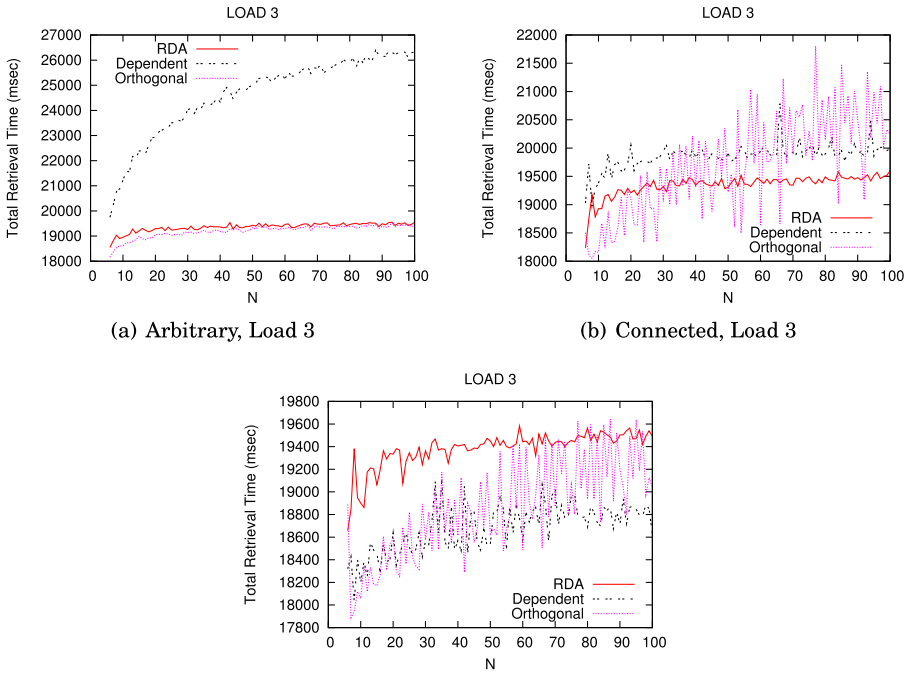
(c) Range, Load 3

Fig. 11. Experiment 2, total retrieval time.

Online returns the optimal retrieval values for experiments 1 through 5, which can be observed from Figure 13 for Experiment 5. This result is not surprising because these experiments are performed for a single site using a single copy for each bucket. When there is a single copy in the system, the retrieval choice of a bucket is obvious. In this case, there is no need to construct the flow graph and to compute the maximum flow. The optimal retrieval schedule from a single site can be calculated using the *Online* algorithm proposed in Algorithm 5, in $O(|Q|)$ operations.

When there is more than one site in the system, then the performance of *Online* gets closer to the optimal values for the experiments that behave like a single site retrieval such as Experiment 7 shown in Figure 14. However, when the disks become more heterogeneous and the initial load/network delay values become more random, as in Experiment 20, then the performance difference between *Online* and the optimal values increases gradually as can be seen in Figure 15.

6.6.4. Running Times of the Proposed Algorithms. In this section, we compare the *Avg. Runtime Per Query* values of the three algorithms proposed; *Max-flow* with binary capacity scaling, *Max-flow* without binary capacity scaling, and *Online*. The results are provided in Figure 16 for Experiment 20. The figure shows three runtime values for every algorithm, each value representing a different allocation scheme. These values are plotted using the same line style. The *Avg. Runtime Per Query* values of different allocation schemes are similar to each other for all the algorithms proposed. A more important factor in the execution time is the heterogeneity of the system. The *Avg. Runtime Per Query* value decreases as the disks become more homogeneous and the load/delay values among the disks become more similar to each other. Since Experiment 20 is the one with all random system parameters, values shown in



(a) Arbitrary, Load 3

(b) Connected, Load 3

(c) Range, Load 3

Fig. 12. Experiment 9, total retrieval time.

Figure 16 are the maximum execution times achieved among the all experiments conducted.

Use of the binary capacity scaling algorithm decreases the execution time of *Max-flow* significantly. For $N = 100$, expected bucket size of a connected query is around 5000 for load 2. While *Max-flow* without the binary capacity scaling algorithm requires 1 to 2 seconds to return the optimal retrieval schedule of 5000 buckets from 100 disks, *Max-flow* with the binary capacity scaling algorithm represented in Algorithm 4 requires only 30 to 70 ms, or 10 microseconds for each bucket on average. *Online* algorithm represented in Algorithm 5 requires less than a millisecond for the retrieval of 5000 buckets from 100 disks by not guaranteeing the optimality of the result.

Execution time of the *Max-flow* approach is highly dependent on the amount of maximum flow calculations performed by the algorithm. Algorithm 4 decreases the execution time significantly since the maximum flow calculations are performed fewer times thanks to the binary capacity scaling algorithm. We investigated the number of max-flow calls performed with and without the binary scaling algorithm and the results are provided in Figure 17 for Experiment 20. Use of the binary capacity scaling algorithm clearly limits the number of max-flow calls to around 10, as can be observed from the figure. This means that the expected number of maximum flow calls is much fewer than the worst case value of $O(log(|Q|) + N)$ presented in Section 5.4.

Execution time of Algorithm 4 also depends on the initial r value chosen for the calculation of the t_{mid} presented in Algorithm 3, line 14; where $t_{mid} = t_{min} + (t_{max} - t_{min}) * r$. Figure 18 shows the position of the optimal retrieval time value within the initial $[t_{min}, t_{max}]$ range for Experiment 9. The Y-axis shows the optimality value calculated using the formula $\frac{opt - t_{min}}{t_{max} - t_{min}}$. It is clear from the figure that the optimum

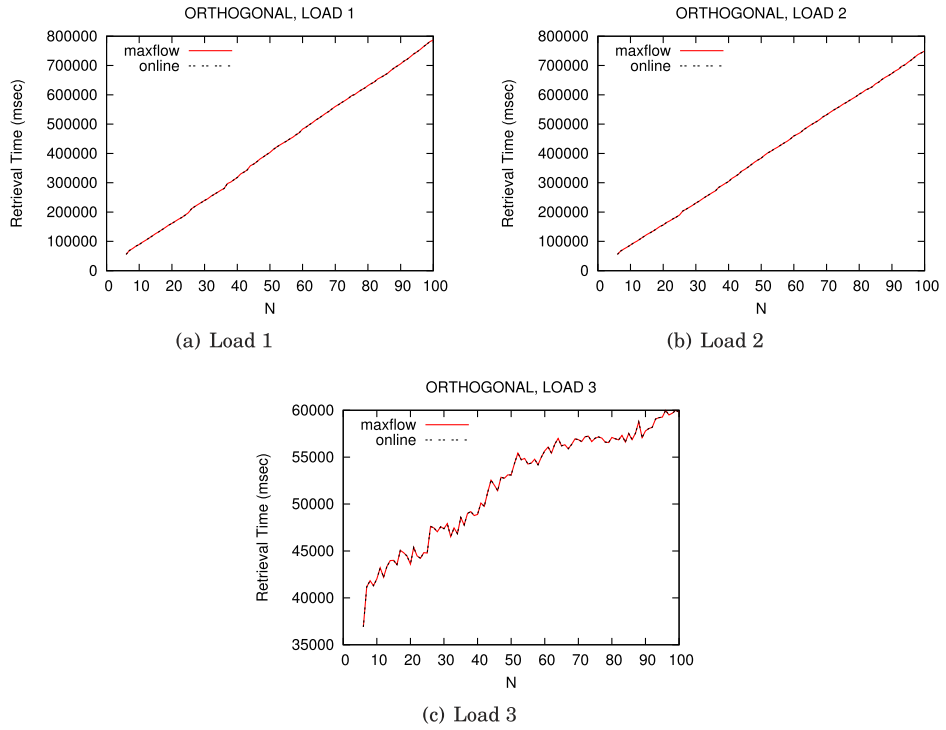
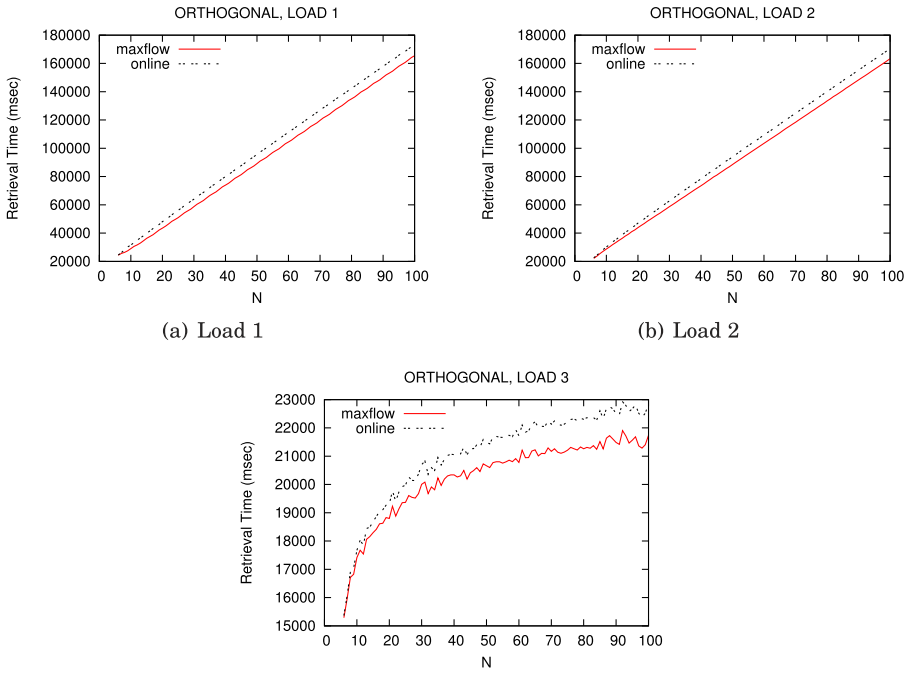


Fig. 13. Experiment 5, *Online* vs. *Max-flow*, arbitrary queries, orthogonal.

retrieval time is much closer to the t_{min} value than the t_{max} value. Choosing an r value closer to the optimality curve presented in the figure plays an important role in shrinking the range faster and consequently decreasing the number of max-flow calls. Note that this r value is needed until the first time t_{max} is decreased, since after that point, the optimal response time can lie anywhere on the range. After the t_{max} is decreased for the first time, we can set r to 0.5 to calculate the middle value in the range. Since the curve in Figure 18(c) is a good upper bound for all the experiments we conducted, we set r to $\frac{2.5}{N}$, which is the equation of the fitted curve in Figure 18(c).

6.6.5. Comparison with Other Approaches. In this section, we compare the performance of the proposed algorithms with the following state of the art approaches.

—*Algorithm-B.* Proposed by Chen and Rotem [1994]. This is the only approach besides ours that we are aware of guaranteeing optimal response time retrieval. The problem is formulated as a network flow problem and a Ford-Fulkerson based algorithm is proposed for the solution guaranteeing the optimality of the result. The limitation of the algorithm is that it can only handle the basic retrieval problem, where the disks within or among the storage arrays are assumed to be homogeneous and the initial loads or the network delays of the disks are not considered.
—*Power of Two.* A heuristic-based solution inspired by Mitzenmacher [2001] for the retrieval of replicated data. In order to retrieve a bucket, two candidate disks are randomly selected among the pool of the disks having a copy of the bucket. Retrieval is performed from the disk resulting in the fastest retrieval time. This decision can be made online considering the disk parameters, initial load of the disk, and the network delay to the site where the disk resides. Actually, our



(a) Load 1

(b) Load 2

(c) Load 3

Fig. 14. Experiment 7, *Online* vs. *Max-flow*, arbitrary queries, orthogonal.

proposed online retrieval algorithm reduces to the *Power of Two* when the number of sites are two. This heuristic does not guarantee the optimality of the result; however, it is expected to perform well considering the low execution time.

— *Random*. Another heuristic-based solution based on a completely random selection. Retrieval of a bucket is performed from the disk chosen randomly among the pool of disks having a copy of the bucket.

First, we compare the performance of the algorithms guaranteeing optimal response time retrieval. Since these algorithms guarantee the optimality of the result, the *Total Retrieval Time* values returned by these algorithms will be the same. Therefore, a better metric for the comparison is looking at the execution time of the algorithms. Here, we compare the *Avg. Runtime Per Query* values returned by our *Max-flow* approach (*mf w/binary*) and *Algorithm-B*. Among the experiments we performed (see Table IV), *Algorithm-B* can only solve Experiment 1 and Experiment 6 because of its limitation of handling only the basic retrieval problem. Since there is no replication in the Experiment 1, we make the comparison for Experiment 6. Note that the proposed algorithms handling the generalized retrieval problem can also handle the basic retrieval problem. Figure 19 shows the *Avg. Runtime Per Query* values returned by *mf w/binary* and *Algorithm-B*. The results are shown for *RDA* using different query types and query loads. Execution times of the algorithms are similar to each other for *Load 3* since the query sizes are smaller in *Load 3*. However, it is clear from the figure that *Algorithm-B* cannot scale well when the request size increases. For *Load 1* and *Load 2*, *Algorithm-B* is up to 40 times slower than *Max-flow*. Performance degradation of *Algorithm-B* is mainly caused by the edge reversals and the extensive number of depth-first searches performed by the Ford-Fulkerson-based solution.

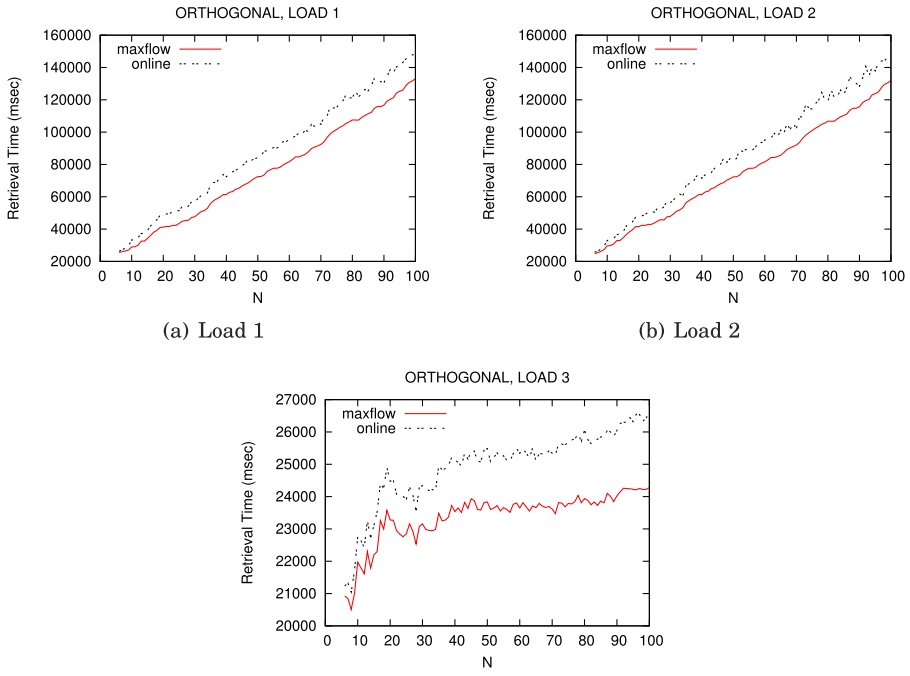
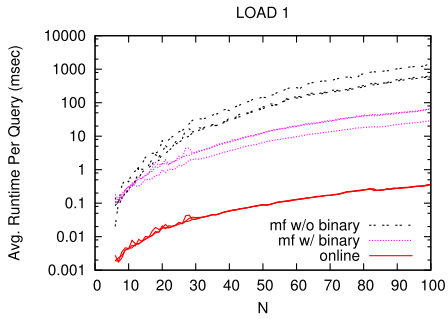


Fig. 15. Experiment 20, *Online* vs. *Max-flow*, arbitrary queries, orthogonal.

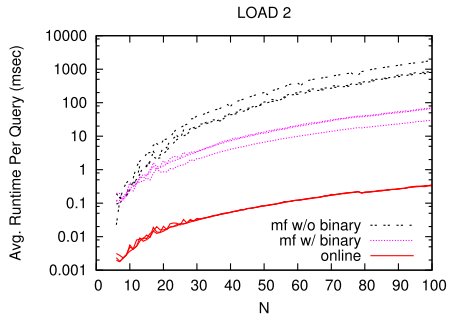
Next, we compare the performance of the approaches that cannot guarantee the optimality of the result. We also plot the optimal retrieval value that our *Max-flow* approach returns, to point out how the other algorithms perform compared to the optimal. Figure 20(a) shows the *Total Retrieval Time* values for Experiment 20 using *RDA* and *Load 1* using *Arbitrary* queries. We collected the results for the number of sites shown on the x-axis using 100 disks per site. As expected, *Power of Two* and *Online* return the same *Total Retrieval Time* values when there are two sites in the system. However, when the number of sites increases, *Online* performs better than *Power of Two*. This is expected since *Power of Two* considers only two sites while *Online* considers all the sites for the fastest retrieval. *Random* performs worst among the three. Even though *Online* seems really close to the optimal in this figure, according to a more detailed comparison made in Figure 15 for two sites, the optimal is still around 1.2 times better than *Online*.

Figure 20(b) compares the execution times of *Online*, *Power of Two*, and *Random*. Execution times of all three approaches are really close to each other; *Random* being the fastest and *Power of Two* being the slowest until the number of sites reaches four. After four sites, *Power of Two* becomes faster than *Online*. Execution times of *Random* and *Power of Two* are more stable when the number of sites increases compared to *Online*. Execution time of *Online* increases linearly as the number of sites increases. This is expected since *Online* considers all the sites in the system for the fastest retrieval of a bucket. We believe that this does not constitute a problem, since the number of sites cannot be very high considering the replication cost of data.

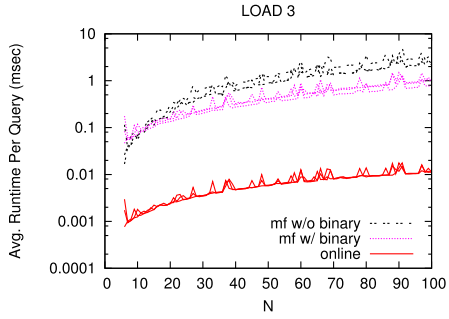
In Figure 21, we compare the performance of *Online*, *Power of Two*, and *Random*, using five sites, different query loads, and query types. In this case, the x-axis shows the number of disks per site. Its clear from the figure that the performance difference



(a) Arbitrary, Load 1



(b) Connected, Load 2

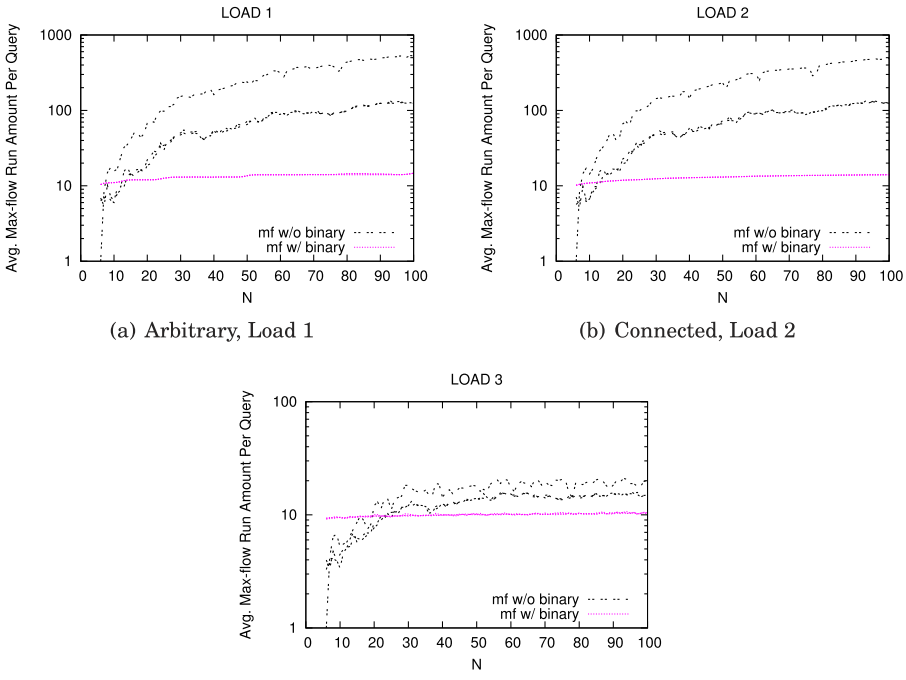


(c) Range, Load 3

Fig. 16. Experiment 20, running time.

between the algorithms does not depend on query type. The retrieval time difference among the algorithms seems constant for *Load 3* shown in Figure 20(c) since the maximum query load (number of buckets requested in a query) is around 150 buckets for *Load 3*. However, it is clear that the retrieval time difference increases as the query load increases. The retrieval time difference between *Power of Two* and *Online* reaches 50 seconds when the query load becomes 5000 for 100 disks of *Load 1* and *Load 2* shown in Figure 21(a) and Figure 21(a) respectively. This difference is more than 3 minutes for *Online* and *Random*. On the other hand, it is not possible to observe such a difference in the execution time of the algorithms. Even for the query load of 5000 buckets for 100 disks shown in Figure 21(d), execution times of the algorithms are similar to each other, all being less than half a millisecond. In summary, *Random* and *Power of Two* cannot recover their retrieval time disadvantage using their execution time since *Online* runs as fast as *Random* and *Power of Two*.

Finally, we compare the performance of different retrieval algorithms using a real-world workload. As a real-world workload, we use the *Exchange* workload explained in Section 6.3. In this experiment, we use the first 15 minute interval of the whole workload covering 24 hours. In this first 15 minute interval used, a total of 236,183 queries (requests) are performed. Minimum query load is 1 bucket and the maximum query load is 1664 buckets, 30 buckets being the mean. There are 9 disks in the system and we use a 100×100 grid to represent all the buckets referenced in the trace. Since the location of copies cannot be retrieved from the trace information, we use *RDA* as the allocation scheme. The results are shown in Figure 22 for the arbitrary queries of Experiment 20. The x-axis shows the number of sites used. Since *Algorithm-B* cannot handle the generalized retrieval problem, results cannot be shown for *Algorithm-B*.



(a) Arbitrary, Load 1

(b) Connected, Load 2

(c) Range, Load 3

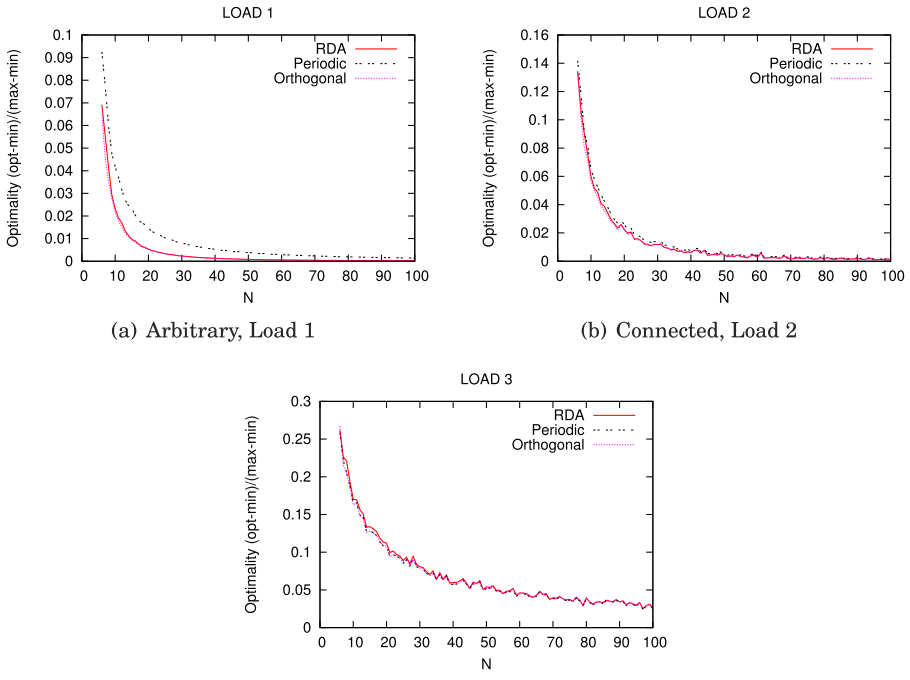
Fig. 17. Experiment 20, max-flow run amount.

First of all, performance of the retrieval algorithms under a real world workload is very similar to that under the synthetic workload shown in Figure 20. This means that the synthetic workloads we used are good representatives of the real world. Second, retrieval time difference between, *Online* and *Max-flow* is as small as 45 seconds when the number of sites reaches five, while this difference was around 6 minutes for two sites. However, the execution time difference between these algorithms is also very low, being less than 0.17 msec per query at most. In fact, when we add the total execution time and the total retrieval time values, *Max-flow* is still the fastest algorithm among all four algorithms for the *Exchange* trace.

6.7. Discussion and Future Work

According to the problem we focus on, we are given a set of buckets to be retrieved and we are asked to find the optimal response time retrieval schedule in the generalized case. For this reason, we focus on one single query (disk request) at a time, where each query is composed of multiple buckets (disk blocks) to be retrieved. Our proposed optimal solutions guarantee that the retrieval schedule yields minimum retrieval time for that query. However, handling the queries in a first-come first-serve fashion might not necessarily result in the minimum total time to finish all the queries even though the retrieval time of each individual query is optimal. Therefore, the next problem would be minimizing the total response time in the general case considering all the queries. Clearly, it is not possible to solve this problem without having a solution to the generalized retrieval problem described in this article: finding the optimal response time retrieval schedule given a set of buckets to be retrieved.

Beside the necessity of the solution proposed in this article, there are multiple trade-offs that need to be considered to find the optimal retrieval time for the total number



(a) Arbitrary, Load 1

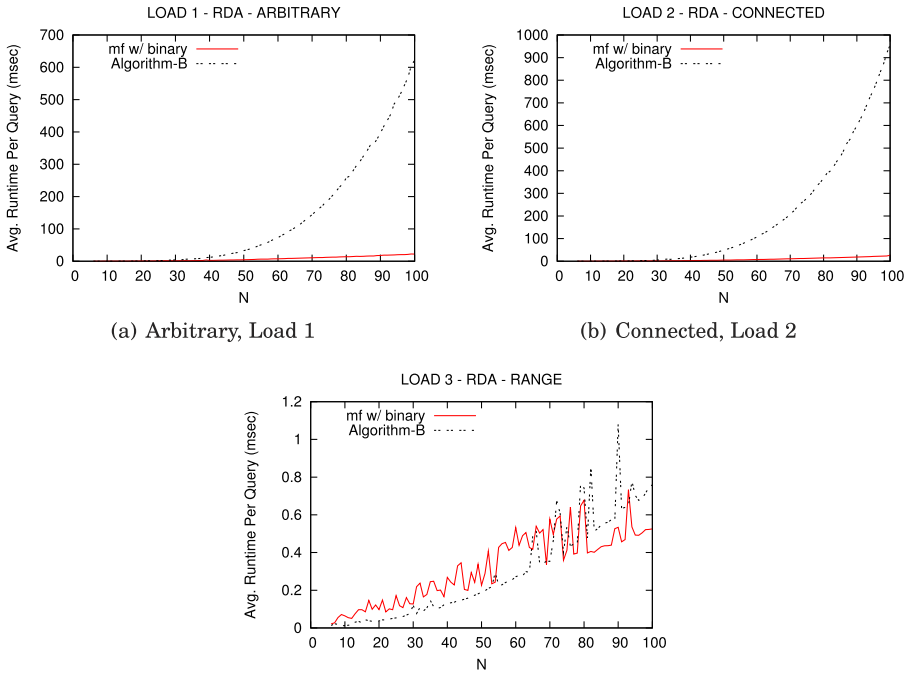
(b) Connected, Load 2

(c) Range, Load 3

Fig. 18. Experiment 9, position of the optimal response time within $[t_{min}, t_{max}]$.

of queries in the generalized case. First of all, we should consider the execution time of the retrieval algorithms and decide between choosing a faster algorithm with a slower retrieval time versus a slower algorithm with a faster retrieval time. This decision might depend on multiple factors that we considered in this article, such as the query load (number of buckets in the query), the query type (*Range*, *Arbitrary*, *Connected* etc.), and the allocation scheme in use (*RDA*, *Dependent Periodic*, *Orthogonal* etc.). Another trade-off to be considered is batching multiple requests and retrieving the buckets all together versus retrieving the buckets of the queries one query at a time as soon as they arrive. This is a harder decision compared to the first. First of all, it is not easy to know when to stop batching and finding the retrieval schedule of the batched queries. While increasing the batch size might be better for load balancing of the disks, causing a better retrieval time, batching too many requests might create a fairness issue by overly delaying some requests. Second, even after stopping the batching process and starting the retrieval of batched queries, it might be better to stop the retrieval at some point and come up with another retrieval decision for the buckets that are left from the previous retrieval, plus the ones that are newly arrived. Another fairness issue might be encountered if some buckets are again overly delayed. As a first follow-up, we are planning to minimize the total response time of the system by considering these trade-offs.

Other interesting future work we are planning to consider, is decreasing the execution time of the *Max-flow* approach proposed in this article. The advantage of the *Max-flow* approach is that it guarantees the optimal response time retrieval schedule for a given query. Thanks to the binary capacity scaling technique, we could decrease the execution time of the *Max-flow* approach to ~50 msec for the retrieval of ~5000 buckets, requiring ~10 microseconds of execution time per bucket. As a

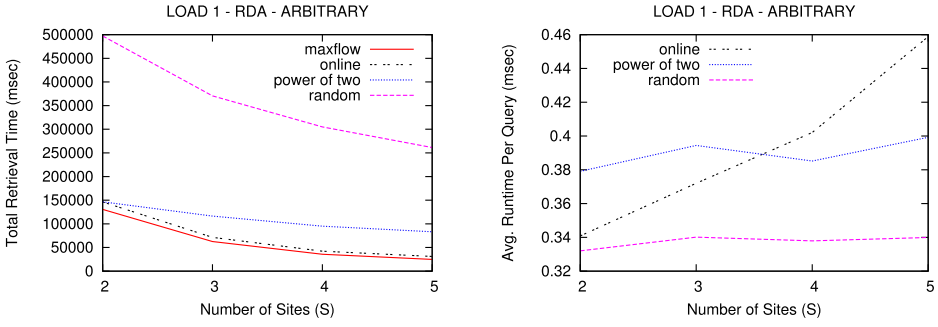


(a) Arbitrary, Load 1

(b) Connected, Load 2

(c) Range, Load 3

Fig. 19. Experiment 6, *Algorithm-B* vs. *mf w/binary*, RDA.

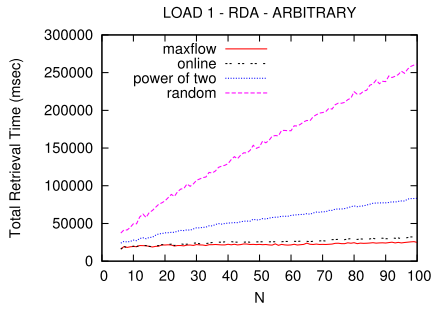


(a) Retrieval Time

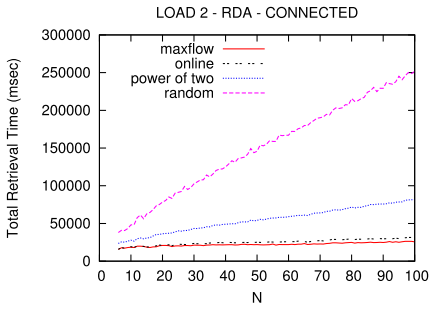
(b) Execution Time

Fig. 20. Experiment 20, Arbitrary, Different Number of Sites, RDA, Load 1.

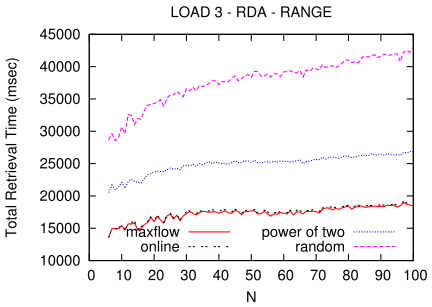
result of the experiments we performed, we found that the *Max-flow* approach yields the smallest total time (retrieval+execution) for all queries compared to the other algorithms and heuristics under the real-world workload of Microsoft's *Exchange* trace (see Figure 22). Since deciding the retrieval schedule is a time-critical issue, as additional future work; we are planning to improve the execution time of the *Max-flow* approach even further. In order to do that, we will first focus on integrated maximum flow algorithms instead of using the maximum flow as a black box technique. Several different implementations of maximum flow can be considered here, such as Ford-Fulkerson-based or Push-relabel-based implementations. Second, since current storage arrays are powered with multicore processors, a multithreaded integrated



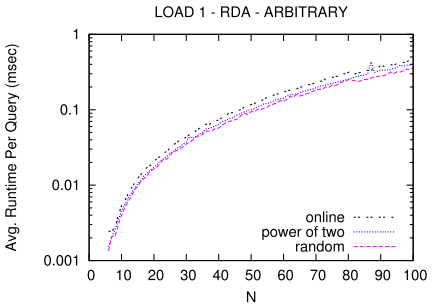
(a) Retrieval Time, Arbitrary, Load 1



(b) Retrieval Time, Connected, Load 2

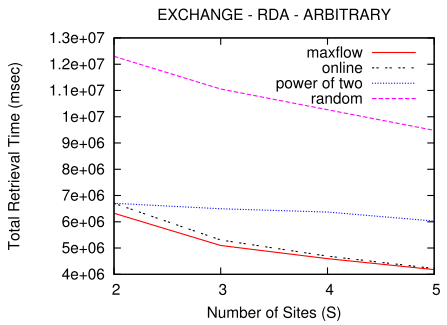


(c) Retrieval Time, Range, Load 3

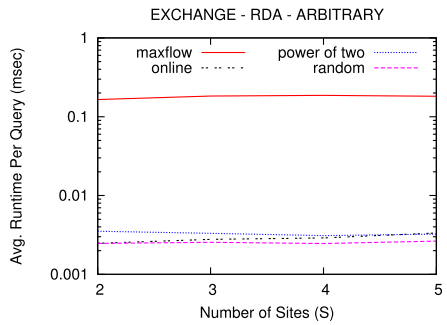


(d) Execution Time, Arbitrary, Load 1

Fig. 21. Experiment 20, 5 Sites, RDA, Synthetic Loads.



(a) Retrieval Time



(b) Execution Time

Fig. 22. Experiment 20, Arbitrary, Different Number of Sites, RDA, Load of the Exchange Trace.

maximum flow solution for the generalized retrieval problem might decrease the execution time even further. There are multiple places where parallelization can be applied. First of all, parallelization can take a place in the maximum flow calculation itself where several parallel maximum flow algorithms have been proposed [Anderson and Setubal 1992; Bader and Sachdeva 2005; Goldberg and Trujan 1988; Hong and He 2011]. Second, parallelization can be performed in the binary capacity scaling technique such that multiple threads might calculate the maximum flow of different subranges of the whole range in parallel, to shrink the range faster.

In summary, we believe that this article is an important milestone for solving many interesting problems of new distributed, heterogeneous storage systems, and it opens the way to interesting future work in this area of research.

7. CONCLUSION

In this article, we investigate the retrieval of declustered data from storage arrays and we allow disks to be heterogeneous, to have initial loads, and network delays. We show how to compute optimal response time retrieval and formulate the problem using LP and maximum flow approaches. We also propose a low complexity online algorithm to compute a retrieval schedule that does not guarantee the optimality of the result. Experimental results using various replication schemes, query types, query loads, disk specifications, site delays, and initial disk loads show that the replication scheme used is important. Orthogonal allocation performs well in many scenarios. The max-flow approach runs a few orders of magnitude faster than the LP model on average and they both yield the optimal response time retrieval. Online algorithm has the fastest execution time; however, it is generally slower than the max-flow approach when we consider its retrieval performance.

REFERENCES

Abdel-Ghaffar, K. A. S. and El Abbadi, A. 1997. Optimal allocation of two-dimensional data. In *Proceedings of ICDT*. 409–418.

Adaptec. 2010. Adaptec high-performance hybrid arrays (HPHAs). http://www.adaptec.com/nr/rdonlyres/a1c72763-e3b9-45f7-b871-a490c29a9b11/0/hpha5_fb.pdf. PMC-Sierra, Inc.

Agrawal, N., Prabhakaran, V., Wobber, T., Davis, J. D., Manasse, M., and Panigrahy, R. 2008. Design tradeoffs for SSD performance. In *Proceedings of Usenix Annual Technical Conference (ATC)*. 57–70.

Altiparmak, N. and Tosun, A. S. 2012. Equivalent disk allocations. *IEEE Trans. Parallel Distrib. Syst. 23*, 3, 538–546.

Anderson, R. J. and Setubal, J. A. C. 1992. On the parallel implementation of Goldberg's maximum flow algorithm. In *Proceedings of the 4th Annual ACM Symposium on Parallel Algorithms and Architectures* (*SPAA*). 168–177.

Atallah, M. J. and Prabhakar, S. 2000. (Almost) optimal parallel block access for range queries. In *Proceedings of ACM PODS*. 205–215.

Bader, D. A. and Sachdeva, V. 2005. A cache-aware parallel implementation of the push-relabel network flow algorithm and experimental evaluation of the gap relabeling heuristic. In *Proceedings of ISCA PDCS*. 41–48.

Beckmann, N., Kriegel, H., Schneider, R., and Seeger, B. 1990. The R* tree: An efficient and robust access method for points and rectangles. In *Proceedings of ACM SIGMOD*. 322–331.

Bhatia, R., Sinha, R. K., and Chen, C. 2000. Hierarchical declustering schemes for range queries. In *Proceedings of EDBT*. 525–537.

Chen, C. and Cheng, C. T. 2002. From discrepancy to declustering: Near optimal multidimensional declustering strategies for range queries. In *Proceedings of ACM PODS*. 29–38.

Chen, C.-M. and Cheng, C. 2003. Replication and retrieval strategies of multidimensional data on parallel disks. In *Proceedings of the Conference on Information and Knowledge Management (CIKM)*.

Chen, C., Bhatia, R., and Sinha, R. 2000. Declustering using golden ratio sequences. In *Proceedings of ICDE*. 271–280.

Chen, L. T. and Rotem, D. 1994. Optimal response time retrieval of replicated data. In *Proceedings of ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*. 36–44.

CPLEX, I. IBM ilog cplex optimization studio for academics: High-performance software for mathematical programming and optimization. http://www.ilog.com/products/cplex/.

Dantzig, G. B. and Thapa, M. N. 1997. *Linear Programming 1: Introduction*. Springer-Verlag.

Du, H. C. and Sobolewski, J. S. 1982. Disk allocation for Cartesian product files on multiple-disk systems. *ACM Trans. Datab. Syst. 7,* 1, 82–101.

EqualLogic. 2011. Equallogic ps6100xs hybrid storage array. http://www.equallogic.com/products/default.aspx?id=10653. Dell, Inc.

Faloutsos, C. and Bhagwat, P. 1993. Declustering using fractals. In *Proceedings of the 2nd International Conference on Parallel and Distributed Information Systems*. 18–25.

Fan, C., Gupta, A., and Liu, J. 1994. Latin cubes and parallel array access. In *Proceedings of the 8th International Parallel Processing Symposium*.

Ferhatosmanoglu, H., Tosun, A. S., and Ramachandran, A. 2004. Replicated declustering of spatial data. In *Proceedings of 23rd ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*. 125–135.

Frikken, K. 2005. Optimal distributed declustering using replication. In *Proceedings of the 10th ICDT*. 144–157.

Frikken, K., Atallah, M., Prabhakar, S., and Safavi-Naini, R. 2002. Optimal parallel I/O for range queries through replication. In *Proceedings of the 13th International Conference on Database and Expert Systems Applications (DEXA)*. 669–678.

Gaede, V. and Gunther, O. 1998. Multidimensional access methods. *ACM Comput. Surv. 30*, 170–231.

Ghandeharizadeh, S. and De Witt, D. J. 1990a. Hybrid-range partitioning strategy: A new declustering strategy for multiprocessor database machines. In *Proceedings of VLDB*. 481–492.

Ghandeharizadeh, S. and De Witt, D. J. 1990b. A multiuser performance analysis of alternative declustering strategies. In *Proceedings of ICDE*. 466–475.

Goldberg, A. V. and Tarjan, R. E. 1988. A new approach to the maximum flow problem. *J. ACM 35*, 921–940.

Guttman, A. 1984. R-trees: A dynamic index structure for spatial searching. In *Proceedings of ACM SIGMOD*. 47–57.

Hong, B. and He, Z . 2011. An asynchronous multithreaded algorithm for the maximum network flow problem with nonblocking global relabeling heuristic. *Trans. Parallel Distrib. Syst 22*, 6, 1025–1033.

Hua, K. A. and Young, H. C. 1997. A general multidimensional data allocation method for multicomputer database systems. In *Proceedings of Database and Expert System Applications*. 401–409.

Karp, R. M. 1972. Reducibility among combinatorial problems. *Complex. Comput. Comput. 40*, 4, 85–103.

Kavalanekar, S., Worthington, B., Zhang, Q., and Sharda, V. 2008. Characterization of storage workload traces from production windows servers. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC)*. 119 –128.

Kim, K. and Prasanna-Kumar, V. K. 1993. Latin squares for parallel array access. *Trans. Parallel Distrib. Syst 4*, 4, 361–370.

Kim, M. H. and Pramanik, S. 1988. Optimal file distribution for partial match retrieval. In *Proceedings of ACM SIGMOD*. 173–182.

Koyuturk, M. and Aykanat, C. 2005. Iterative-improvement-based declustering heuristics for multi-disk databases. *Inform. Syst. 30,* 9, 47–70.

Liu, D. and Wu, M. 2001. A hypergraph based approach to declustering problems. *Distrib. Parallel Datab. 10*, 3.

Mehlhorn, K. and Näher, S. 1995. Leda: A platform for combinatorial and geometric computing. *Comm. ACM 38*, 1, 96–102.

Mitzenmacher, M. 2001. The power of two choices in randomized load balancing. *Trans. Parallel Distrib. Syst. 12*, 1094–1104.

Narayanan, D., Donnelly, A., Thereska, E., Elnikety, S., and Rowston, A. 2008. Everest: Scaling down peak loads through I/O off-loading. In *Oper. Syst. Design Implement*. 15–28.

Narayanan, D., Thereska, E., Donnelly, A., Elnikety, S., and Rowston, A. 2009. Migrating server storage SSDs: Analysis and tradeoffs. In *Proceedings of EuroSystems*. 145–158.

Nimbus. 2010. Nimbus data s-class enterprise flash storage systems. http://www.nimbusdata.com/products/Nimbus_S-class_Datasheet.pdf.

Oktay, K. Y., Turk, A., and Aykanat, C. 2009. Selective replicated declustering for arbitrary queries. In *Proceedings of the 15th International Euro-Par Conference on Parallel Processing. (Euro-Par)*. Springer-Verlag, Berlin, 375–386.

Orenstein, G. 2003. *IP Storage Networking: Straight to the Core*. Addison-Wesley.

Prabhakar, S., Abdel-Ghaffar, K., Agrawal, D., and El Abbadi, A. 1998a. Cyclic allocation of two-dimensional data. In *Proceedings of ICDE*. 94–101.

Prabhakar, S., Agrawal, D., and El Abbadi, A. 1998b. Efficient disk allocation for fast similarity searching. In *Proceedings of SPAA*. 78–87. PW.

Ramsan. 2010. Ramsan-630 flash solid state disk. http://www.ramsan.com/files/download/212. White Paper, Texas Memory Systems.

Samet, H. 1989. *The Design and Analysis of Spatial Structures.* Addison Wesley, MA.

Sanders, P., Egner, S., and Korst, K. 2000. Fast concurrent access to parallel disks. In *Proceedings of the 11th ACM-SIAM Symposium on Discrete Algorithms.*

Shektar, S. and Liu, D. 1996. Partitioning similarity graphs: A framework for declustering problems. *Inform. Syst. 21*, 4.

SNIA. Iotta repository. http://iotta.snia.org. Storage Networking Ind. Assoc.

Sun. 2009a. Sun storage 7000 unified storage systems family. http://www.oracle.com/us/products/servers-storage/039224.pdf.

Sun. 2009b. Sun storage f5100 flash array. http://www.oracle.com/us/043970.pdf.

Tosun, A. S. 2004. Replicated declustering for arbitrary queries. In *Proceedings of the 19th ACM Symposium on Applied Computing*. 748–753.

Tosun, A. S. 2005a. Constrained declustering. In *Proceedings of the International Conference on Information Technology Coding and Computing.* 232–237.

Tosun, A. S. 2005b. Design theoretic approach to replicated declustering. In *Proceedings of the International Conference on Information Technology Coding and Computing.* 226–231.

Tosun, A. S. 2005c. Threshold based declustering in high dimensions. In *Proceedings of the International Conference on Database and Expert Systems Applications*. 818–827.

Tosun, A. S. 2007a. Analysis and comparison of replicated declustering schemes. *Trans. Parallel Distrib. Syst. 18*, 11, 1578–1591.

Tosun, A. S. 2007b. Threshold-based declustering. *Inform. Sci. 177*, 5, 1309–1331.

Tosun, A. S. 2008. Multi-site retrieval of declustered data. In *Proceedings of 28th International Conference on Distributed Computing Systems (ICDCS).* 486–493.

Tosun, A. S. and Ferhatosmanoglu, H. 2002. Optimal parallel I/O using replication. In *Proceedings of International Workshops on Parallel Processing (ICPP).* 506–513.

Violin. 2010. Violin 3200 flash memory array. http://www.violin-memory.com/assets/3200-datasheet.pdf. Violin 3200 Memory Datasheet.

Violin. 2011. Violin 6000 flash memory array. http://www.violin-memory.com/assets/Violin_Datasheet_6000.pdf?d=1. Violin 6000 Memory Datasheet.

XO. Dedicated Internet access overview. http://www.xo.com/services/network/dia/Pages/overview. aspx. XO Communications, LLC.

Zebi. 2012. Zebi hybrid storage array. http://tegile.biz/wp-content/uploads/2012/01/Zebi-White-Paper-012612-Final.pdf. Tegile Systems, Inc.

Received May 2012; revised September 2012; accepted December 2012