Available at
www.ElsevierComputerScience.com
POWERED BY SCIENCE DIRECT®

Information Systems 30 (2005) 47–70

www.elsevier.com/locate/infosys

Iterative-improvement-based declustering heuristics for multi-disk databases[☆], [☆☆]

Mehmet Koyutürk[a,1], Cevdet Aykanat[b,*]

[a] Department of Computer Sciences, Purdue University, West Lafayette, IN 47907, USA
[b] Computer Engineering Department, Bilkent University, Ankara 06800, Turkey

Received 10 July 2002; received in revised form 21 May 2003; accepted 29 August 2003

Abstract

Data declustering is an important issue for reducing query response times in multi-disk database systems. In this paper, we propose a declustering method that utilizes the available information on query distribution, data distribution, data-item sizes, and disk capacity constraints. The proposed method exploits the natural correspondence between a data set with a given query distribution and a hypergraph. We define an objective function that exactly represents the aggregate parallel query-response time for the declustering problem and adapt the iterative-improvement-based heuristics successfully used in hypergraph partitioning to this objective function. We propose a two-phase algorithm that first obtains an initial K-way declustering by recursively bipartitioning the data set, then applies multi-way refinement on this declustering. We provide effective gain models and efficient implementation schemes for both phases. The experimental results on a wide range of realistic data sets show that the proposed method provides a significant performance improvement compared with the state-of-the-art declustering strategy based on similarity-graph partitioning.

Keywords: Parallel database systems; Declustering; Hypergraph partitioning; Iterative improvement; Weighted similarity graph; Max-cut graph partitioning

1. Introduction

Minimizing query-response times is a crucial issue in designing high-performance database systems for application domains such as scientific, spatial and multimedia. These systems are often used interactively and amounts of data to be retrieved for individual queries are quite large. In such database systems, the I/O bottleneck is overcome through parallel I/O across multiple disks. Disks are accessed in parallel while processing a query, so response time for a query can be

[☆] Recommended by K.A. Ross.

[☆☆] This work was partially supported by The Scientific and Technical Research Council of Turkey under grant 199E013.

[*] Corresponding author. Department of Computer Engineering, Bilkent University, Ankara 06533, Turkey. Tel.: +90-312-290-1625 fax: +90-312-266-4047.

E-mail addresses: koyuturk@cs.purdue.edu (M. Koyutürk), aykanat@cs.bilkent.edu.tr (C. Aykanat).

[1] The author was an M.Sc. student at the Computer Engineering Department of Bilkent University during this work.

0306-4379/$ - see front matter © 2003 Elsevier Ltd. All rights reserved.
doi:10.1016/j.is.2003.08.003

minimized by balancing the amount of data to be retrieved on each disk. Therefore, data is distributed across multiple disks, respecting disk capacity constraints, in such a way that data items that are more likely to be retrieved together are located into separate disks. This operation is known as declustering.

There have been considerable amounts of research on developing strategies to effectively decluster data on several disks in order to achieve minimum I/O cost. Many declustering strategies were developed on declustering multi-dimensional data structures such as cartesian product files, grid files, quad trees and R-trees [1–8], multimedia databases [9–13], parallel web servers [14], signature files [15], spatial databases and geographic information systems (GIS) [16,17].

In the literature, there exists a vast amount of work on mapping-function-based declustering techniques such as Coordinate Modulo Declustering (CMD) [5], Field-wise Exclusive-OR Distribution [18], Hilbert Curve Method [3,19], Lattice Allocation Method [2], and Cyclic Allocation Scheme [8]. Commonly these methods scatter the data into disks in such a way that the neighboring data items in multi-dimensional space are placed into different disks. The applications of these methods are restricted to spatial databases and multi-attribute data sets. Furthermore, if there exists information about query distribution and data sizes, these methods do not exploit such available information.

Recently, Shekhar and Liu [16] proposed a novel declustering technique which can exploit information about query distribution and handle heterogeneous data-item sizes, non-uniform data distributions, and constraints on disk sizes. They model the declustering problem as max-cut partitioning of a weighted similarity graph (WSG). The nodes of WSG correspond to data items and weights associated with edges represent similarity between respective data-item pairs. Here, the similarity between a pair of data items refers to the likelihood that the pair will be accessed together by queries. Hence, maximizing the edge cut in a partitioning of WSG relates to maximizing the chance of assigning similar data items to separate disks. This model was reported [16] to outperform all mapping-function-based strategies in experiments with grid files. In this work, we show that the objective function of max-cut graph partitioning does not accurately represent the cost function of declustering. This flaw is because of the fact that WSG is an indirect model and it represents each query defining a single multi-way relation by multiple pairwise relations.

In this work, we propose a direct model for solving the declustering problem by exploiting the correspondence between a data set with a given query distribution and a hypergraph. Each data item and query in the database system correspond, respectively, to a vertex and a hyperedge (net) of the hypergraph. The hypergraph partitioning (HP) problem has been widely encountered in VLSI layout design [20,21] and partitioning irregular computational domains for parallel computing [22,23]. We define an objective function that exactly represents the total query response time for the declustering problem and adapt the iterative-improvement-based HP algorithms to this objective function. We propose a two phase algorithm that first obtains an initial K-way declustering by recursively bipartitioning the data set, then applies multi-way refinement on this declustering. We provide effective gain models and efficient implementation schemes for both phases. Experimental results on a wide range of realistic data sets show that the proposed model provides significantly better declusterings than the WSG model, which is the most promising strategy in the literature.

We define the declustering problem and introduce the notation in Section 2. In Section 3, we introduce the state-of-the-art WSG model and discuss the flaws of this model. We introduce our model and the adaptation of iterative-improvement techniques to the problem in Section 4. In Section 5, we report the experimental results and evaluate the performance of the proposed method.

2. Basic definitions on declustering

Declustering problem can be defined in various ways depending on the application. Shekhar and Liu [16] define the problem in a database environment with a given data set and a query set. Information on possible queries can be available in many database applications, possible queries may be predicted using the information on the application or queries may be logged with the reasonable assumption that the queries that will be processed in the future will be similar to the recent ones. In some cases, information on queries may not be available and it can be more appropriate to decluster the data in such a way that the data items sharing a common feature are stored on separate disks. This can be the case in some multimedia servers [11,12] or content-based image retrieval systems [24,13]. Therefore, it will be more convenient to provide a definition of the problem in terms of a set of data items and a set of relations among data items as in the work of Zhou and Williams [25]. The set of relations may refer to the query set or a possible query may be the union of a set of relations in many applications.

In the framework of this paper, the declustering problem is defined on a database system represented as a two-tuple $(\mathcal{D}, \mathcal{Q})$. $\mathcal{D}$ is the set of data items, where each data item $d \in \mathcal{D}$ may be a spatial object, a multi-dimensional vector, a signature, or a cluster of records depending on the application. $\mathcal{Q}$ is the set of relations over $\mathcal{D}$, where a relation $q \in \mathcal{Q}$ is defined to be a subset of $\mathcal{D}$ (i.e., $q \subseteq \mathcal{D}$). A relation is a query in applications for which prior information on queries is available. In some other applications, a relation may be a pattern, a spatial neighborhood, or a bit position in a signature file. A query will generally be the union of some relations in such applications. Thus, without loss of generality, we will use the term query instead of relation throughout this paper for convenience.

Queries are associated with a relative frequency function $f: \mathcal{Q} \rightarrow [0,1]$, where $f(q)$ shows the likelihood of processing query q, i.e., the tendency of the data items in q to be accessed together. Data items are associated with two size functions $w, t: \mathcal{D} \rightarrow \mathcal{Z}^+$. Here, $w(d)$ relates to the amount of storage requirement for data item d, and $t(d)$ relates to the retrieval time of d from a disk. In practice, these two size functions are closely related since the I/O time required to retrieve a data item is linearly proportional to its storage size in general. Such relative weighting might be necessary, because the sizes of data items can vary significantly in many database systems like GIS or multimedia applications [16]. In this paper, we will refer to such systems as database systems with heterogeneous data-item sizes. If all data items have equal retrieval times, such systems will be referred to as database systems with homogeneous data-item sizes.

Definition 1. A K-way declustering of $(\mathcal{D}, \mathcal{Q})$ is a K-way partition $\Pi_K = \{\mathcal{D}_1, \mathcal{D}_2, \ldots, \mathcal{D}_K\}$ of $\mathcal{D}$ to K disks, where parts are mutually exhaustive and disjoint (i.e., $\cup_{k=1}^{K} \mathcal{D}_k = \mathcal{D}$ and $\mathcal{D}_k \cap \mathcal{D}_\ell = \emptyset$ for $1 \leqslant k \neq \ell \leqslant K$).

Definition 2. A declustering Π_K is said to be feasible if each part $\mathcal{D}_k$ satisfies a given disk capacity constraint, i.e., $W_k = \sum_{d \in \mathcal{D}_k} w(d) \leqslant C_k$ for $1 \leqslant k \leqslant K$. Here, C_k denotes the capacity of disk D_k.

Definition 3. In a declustering Π_K, response time $r(q)$ for a query q is $r(q) = \max_{1 \leqslant k \leqslant K}\{t_k(q)\}$, where $t_k(q) = \sum_{d \in q \cap \mathcal{D}_k} t(d)$ denotes the total retrieval time of data items on disk D_k that qualify for q. The aggregate parallel response time for a query set $\mathcal{Q}$ is $R(\mathcal{Q}) = \sum_{q \in \mathcal{Q}} f(q) r(q)$.

Definition 4. A declustering Π_K is said to be strictly optimal with respect to a query set $\mathcal{Q}$ if and only if it is optimal for every query $q \in \mathcal{Q}$, i.e., $r(q) = r_{opt}(q)$, for each $q \in \mathcal{Q}$.

The problem of finding an optimal distribution of data items in a single query q into K disks is equivalent to the well-known number partitioning problem which is known to be NP-hard [26]. However, for database systems with homogeneous data-item sizes, $r(q) = \max_{1 \leqslant k \leqslant K}\{|q \cap \mathcal{D}_k|\}$ with the assumption of unit data retrieval time. Thus, this individual problem becomes trivial so that $r_{opt}(q) = \lceil |q|/K \rceil$. The concept of strict optimality refers to attaining maximum parallelism without any overhead due to allocation conflicts among individual queries. We use the term *allocation conflict* for the case when the strictly-optimal declustering of a group of queries enforces non-optimal declustering of at least one query.

Definition 5. In a declustering Π_K, the aggregate parallel response overhead for a query set $\mathcal{Q}$ is $R_O(\mathcal{Q}) = \sum_{q\in\mathcal{Q}} f(q)(r(q) - r_{opt}(q))$.

Definition 6. Given a database system $(\mathcal{D}, \mathcal{Q})$, the declustering problem is defined as finding a feasible K-way declustering Π_K that minimizes the aggregate parallel response time $R(\mathcal{Q})$ which is equivalent to minimizing aggregate parallel response overhead $R_O(\mathcal{Q})$.

The equivalence between these two objective functions can easily be seen as follows:

$$\begin{aligned} R_O(\mathcal{Q}) &= \sum_{q\in\mathcal{Q}} f(q)(r(q) - r_{opt}(q)) \\ &= \sum_{q\in\mathcal{Q}} f(q)r(q) - \sum_{q\in\mathcal{Q}} f(q)r_{opt}(q) \\ &= R(\mathcal{Q}) - R_{opt}(\mathcal{Q}). \end{aligned} \tag{1}$$

Here, $R_{opt}(\mathcal{Q})$ denotes the weighted sum of optimal parallel response times of all queries, which is equal to the aggregate parallel response time of a strictly optimal declustering if it exists. So, $R_{opt}(\mathcal{Q})$ is a constant. Therefore, $R(\mathcal{Q})$ is minimized if and only if $R_O(\mathcal{Q})$ is minimized. $R_O(\mathcal{Q})$ has the nice property of providing information on "how far the declustering is away from being strictly optimal" making the cost equal to zero for a strictly optimal declustering. So, we prefer this metric as the cost function.

3. Flaws of weighted similarity graph (WSG) model

In the model proposed by Shekhar and Liu [16], a database system $(\mathcal{D}, \mathcal{Q})$ is represented by a weighted similarity graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$. In $\mathcal{G}$, $\mathcal{V} \equiv \mathcal{D}$ so that vertex v_i represents data item d_i. Each query $q \in \mathcal{Q}$ is represented by a clique of vertices corresponding to the data items that qualify for q. That is, each query q induces an edge between every pair of vertices representing its qualifying data items. For database systems with homogeneous data-item sizes, each edge in the clique is weighted with the frequency of q. The multiple edges connecting each pair of vertices of $\mathcal{G}$ are contracted into a single edge of which weight is equal to the sum of weights of the edges it represents. Formally, $\mathcal{E} \equiv \{(v_i, v_j) \mid v_i, v_j \in \mathcal{V}$ and $\exists q \in \mathcal{Q} \ni d_i, d_j \in q\}$ with $w(v_i, v_j) = \sum_{q\in\mathcal{Q}_{ij}} f(q)$. Here, $\mathcal{Q}_{ij} \subseteq \mathcal{Q}$ is the set of all queries that contain both d_i and d_j. Then, the problem of declustering $(\mathcal{D}, \mathcal{Q})$ is formulated as max-cut partitioning of $\mathcal{G}$. The max-cut graph partitioning problem is defined as the task of finding a feasible K-way partition $\Pi_K = \{\mathcal{V}_1, \mathcal{V}_2, \ldots, \mathcal{V}_K\}$ of $\mathcal{V}$ that maximizes the cutsize of the partition. The cutsize of Π_K is defined as the sum of weights of cut edges, where an edge is said to be cut if it connects a pair of vertices belonging to two different parts. The max-cut graph partitioning problem is known to be NP-hard [26].

In WSG, edge weights represent the similarity between the end vertices, where the similarity between two data items is defined as the likelihood of being accessed together by queries in $\mathcal{Q}$. So, in the WSG model, maximizing the cutsize is expected to minimize the aggregate parallel response overhead through maximizing the likelihood of assigning pairs of data items that are frequently accessed together to separate disks.

Shekhar and Liu [16] prove that the WSG model is able to find a strictly optimal declustering if it exists for a database system with homogeneous data-item sizes. However, if no strictly optimal declustering exists, the optimal partition for the WSG model may be far away from being an optimal declustering. This flaw follows from the fact that the WSG model lacks proper scaling in resolving allocation conflicts among different queries. This is because multi-item relations defined by individual queries are represented as separate pairwise relations between data items.

Consider two different query subsets $\mathcal{Q}_1 = \{\{d_1, d_2, d_3\}\}$ and $\mathcal{Q}_2 = \{\{d_1, d_2\}, \{d_1, d_3\}, \{d_2, d_3\}\}$ in a database system $(\mathcal{D}, \mathcal{Q})$, where $\mathcal{D} = \{d_1, d_2, d_3\}$ and all queries have equal frequencies. Both query subsets induce the same subgraph in WSG, which is a triangle with equally weighted edges, so contributions of $\mathcal{Q}_1$ and $\mathcal{Q}_2$ to the cutsize will be the same under any given partitioning. However, in a two-way declustering where all three data items are assigned to the same part, we will have considerably different parallel response overheads of $R_O(\mathcal{Q}_1) = (R(\mathcal{Q}_1) - R_{opt}(\mathcal{Q}_1))/|\mathcal{Q}| = (3-2)/|\mathcal{Q}| = 1/|\mathcal{Q}|$ and $R_O(\mathcal{Q}_2) = (R(\mathcal{Q}_2) - R_{opt}(\mathcal{Q}_2))/|\mathcal{Q}| = (6-4)/|\mathcal{Q}| = 2/|\mathcal{Q}|$. In this example, the WSG model overestimates the importance of $\mathcal{Q}_1$ compared to $\mathcal{Q}_2$ in terms of contribution to the cutsize.

In a declustering, the parallel response time for a query q is proportional to the maximum of the number of data items on each disk qualifying for q. So the actual objective should be minimizing the distribution imbalances of all queries. However, in the WSG model, contribution of the clique induced by a query q to the cutsize, referred to as the cutsize due to q, relates to the variance in the distribution of data items in q over disks rather than the imbalance of the distribution. In database systems with allocation conflicts among queries, this flaw in the optimization metric may lead to erroneous situations such that larger cutsizes may correspond to worse parallel response times for individual queries. For example, consider two different 3-way declusterings Π'_3 and Π''_3 with distributions [5:5:1] and [6:3:2], respectively, for a query q of size $|q| = 11$. Here, size of a query q refers to the number of data items that qualify for q. In Π'_3, distribution [5:5:1] for q shows that 5, 5, and 1 data items of q reside on disks D_1, D_2, and D_3, respectively. So, in Π'_3, $r(q) = \max\{5,5,1\} = 5$ and the cutsize due to q is equal to $(5\times 5) + (5\times 1) + (5\times 1) = 35$. Although distribution [6:3:2] for q in Π''_3 incurs a larger (better) cutsize of 36 in WSG, it leads to a larger (worse) parallel response time of 6.

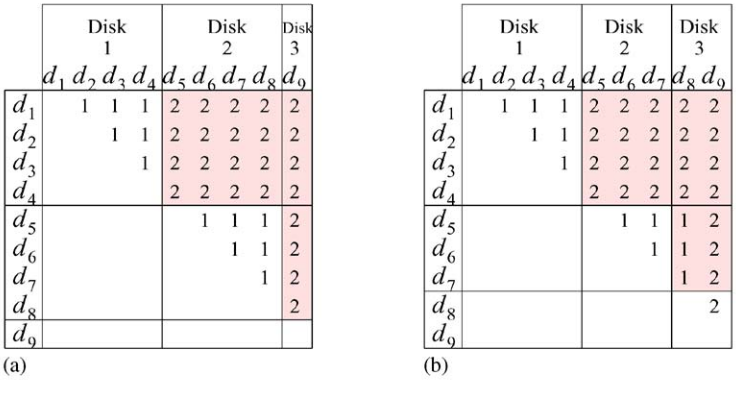
We finalize the discussion for the homogeneous case with a complete example. Fig. 1 shows a database system $(\mathcal{D}, \mathcal{Q})$ with nine data items and 25 queries, and two different 3-way partitions Π'_3 and Π''_3 of the corresponding WSG displayed in adjacency matrix representation. Since WSG is an undirected graph, only the upper triangular portion of its symmetric adjacency matrix is shown. Off-diagonal blocks are colored into grey to show the cut edges of a partition so that the sum of the numbers in grey blocks is equal to the cutsize. Π'_3 shown in Fig. 1(a) is the only optimal declustering for $(\mathcal{D}, \mathcal{Q})$, with aggregate parallel response time of 28, and the cutsize of Π'_3 on WSG is equal to 48. Although Π''_3 shown in Fig. 1(b) is an optimal

$$\mathcal{D} = \{d_1, d_2, \ldots, d_9\} \quad \mathcal{Q} = \{q_1, q_2, \ldots, q_{25}\}$$

$$q_1 = \{d_1, d_2, d_3, d_4, d_5, d_6, d_7, d_8, d_9\}$$

$$q_2, q_3, \ldots, q_{17} = \{d_i, d_j\}, 1 \le i \le 4, 5 \le j \le 8$$

$$q_{18}, q_{19}, \ldots, q_{25} = \{d_i, d_9\}, 1 \le i \le 8$$



(a)

	Disk 1				Disk 2				Disk 3
	d_1	d_2	d_3	d_4	d_5	d_6	d_7	d_8	d_9
d_1		1	1	1	2	2	2	2	2
d_2			1	1	2	2	2	2	2
d_3				1	2	2	2	2	2
d_4					2	2	2	2	2
d_5						1	1	1	2
d_6							1	1	2
d_7								1	2
d_8									2
d_9									

(b)

	Disk 1				Disk 2			Disk 3	
	d_1	d_2	d_3	d_4	d_5	d_6	d_7	d_8	d_9
d_1		1	1	1	2	2	2	2	2
d_2			1	1	2	2	2	2	2
d_3				1	2	2	2	2	2
d_4					2	2	2	2	2
d_5						1	1	1	2
d_6							1	1	2
d_7								1	2
d_8									2
d_9									

Fig. 1. Adjacency matrix representations of WSG of a sample database system with 9 data items of equal size and 25 queries of equal frequency for 3-way declusterings: (a) $\Pi'_3 = \{\{d_1, d_2, d_3, d_4\}, \{d_5, d_6, d_7, d_8\}, \{d_9\}\}$ with cutsize 48 and aggregate parallel response time 28, and (b) $\Pi''_3 = \{\{d_1, d_2, d_3, d_4\}, \{d_5, d_6, d_7\}, \{d_8, d_9\}\}$ with cutsize 49 and aggregate parallel response time 29. (Note: equal query frequencies are ignored for the sake of clarity).

$$\mathcal{D}=\{d_1,d_2,d_3,d_4,d_5\} \qquad \mathcal{Q}=\{q_1,q_2\}$$

$$t(d_1)=t(d_2)=t(d_3)=1,\ t(d_4)=3,\ t(d_5)=5 \quad q_1=\{d_1,d_2,d_5\},\ q_2=\{d_1,d_2,d_3,d_4\}$$

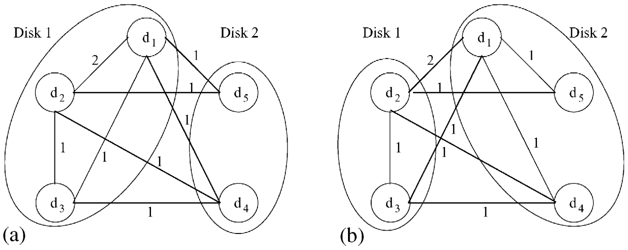


Fig. 2. WSG of a sample database system with 5 data items of different retrieval times and two queries of equal frequency for two-way declusterings: (a) $\Pi'_2 = \{\{d_1, d_2, d_3\}, \{d_4, d_5\}\}$ with cutsize five and aggregate parallel response time 8, and (b) $\Pi''_2 = \{\{d_2, d_3\}, \{d_1, d_4, d_5\}\}$ with cutsize six and aggregate parallel response time 10 (Note: equal query frequencies are ignored in the computation of response times for the sake of clarity).

partition of WSG providing a greater cutsize of 49, it incurs a worse aggregate parallel response time of 29. This discrepancy between the objectives of WSG model and the declustering problem is caused by the allocation conflict between the strictly optimal allocation of query group $\{q_2, q_3, \ldots, q_{25}\}$ and query q_1. The WSG model resolves this conflict by sacrificing optimal allocation of q_{25} for less variance in the distribution of q_1 although this does not reduce the parallel response time for q_1.

For database systems with heterogeneous data-item sizes, Shekhar and Liu [16] scale the weight of each edge (v_i, v_j) of WSG with $\min\{t(d_i), t(d_j)\}$, that is $w(v_i, v_j) = \min\{t(d_i), t(d_j)\} \times \sum_{q \in \mathcal{Q}_{ij}} f(q)$ for each edge $(v_i, v_j) \in \mathcal{E}$. This scaling factor relates to the possible savings in response time achieved by assigning items d_i and d_j to separate disks instead of allocating them into the same disk, i.e., $\min\{t(d_i), t(d_j)\} = (t(d_i) + t(d_j)) - \max\{t(d_i), t(d_j)\}$. Thus, the sum of possible savings in response times is maximized by maximizing the cutsize on WSG. However, the sum of pairwise savings for a query q of size greater than 2 is only a coarse approximation to the actual saving achieved by parallelizing the retrieval of q. For instance, for a query $q = \{d_1, d_2, d_3\}$, with $t(d_1) = 1$, $t(d_2) = 2$, $t(d_3) = 3$, the weights of edges between corresponding vertices are $w(v_1, v_2) = 1$, $w(v_1, v_3) = 1$ and $w(v_2, v_3) = 2$ ignoring the frequency of q. In a two-way declustering $\Pi_2 = \{\mathcal{D}_1 = \{d_1, d_3\}, \mathcal{D}_2 = \{d_2\}\}$, the actual saving is $(t(d_1) + t(d_2) + t(d_3)) - \max\{t(d_1) + t(d_3), t(d_2)\} = 6 - 4 = 2$, whereas the sum of pairwise savings estimated by the WSG model is $\min\{t(d_1), t(d_2)\} + \min\{t(d_3), t(d_2)\} = 1 + 2 = 3$. The WSG model ignores the difference between the retrieval times of data items d_2 and d_3, although the decision of allocating d_2 or d_3 to the same disk with d_1 affects the parallel response time for q.

A sample database system for which the WSG model is unable to find the existing strictly-optimal declustering is shown in Fig. 2. Π'_2 shown in Fig. 2(a) is a strictly-optimal two-way declustering with aggregate parallel response time 8. Π''_2 shown in Fig. 2(b) is an optimal partition for WSG model. Although Π''_2 has a larger cutsize than that of Π'_2, it has a worse (larger) aggregate parallel response time 10.

4. Hypergraph model for declustering

A database system $(\mathcal{D}, \mathcal{Q})$ can naturally be described as a hypergraph. A hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{E})$ is a generalized version of a graph in which each edge $e \in \mathcal{E}$, usually referred to as hyperedge, can connect possibly more than two vertices, i.e., $e \subseteq \mathcal{V}$. So, each hyperedge can naturally represent a query which may define a single relation among more than two data items. That is, in $\mathcal{H}$, $\mathcal{V} \equiv \mathcal{D}$ and $\mathcal{E} \equiv \mathcal{Q}$ so that vertex v_i represents data item d_i

and hyperedge e_j represents query q_j. The vertices connected by hyperedge e_j, referred to as pins of e_j, correspond to the data items that qualify for query q_j. Each hyperedge is associated with a weight equal to the frequency of the respective query.

With this representation, the declustering problem for a database system with homogeneous data-item sizes can be modeled as a hypergraph partitioning (HP) problem with proper modification on the objective function. Traditionally, the HP problem is defined as partitioning the hypergraph into equally weighted parts to minimize the weighted sum of connectivities of hyperedges. Here, connectivity of a hyperedge e refers to the number of parts in which at least one pin of e is allocated. If the size of each query is less than or equal to the number of disks, then the declustering problem can be exactly modeled as a max-cut HP problem, where the objective function corresponds to maximizing the weighted sum of the connectivities of hyperedges. In the general case, the objective is to minimize the weighted sum of the bottleneck values of pin distributions of hyperedges. Here, the bottleneck value of pin distribution of a hyperedge e refers to the number of pins of e in the bottleneck part, which is the part that contains the maximum number of pins of e over all parts.

As the HP problem is known to be NP-hard [21], a vast amount of research has been conducted to develop efficient heuristics and tools for the solution of this well-known problem. Iterative-improvement heuristics introduced by Kernighan–Lin (KL) [27] and Fiduccia-Mattheyses (FM) [28] have been widely used for graph/hypergraph bipartitioning because of their short run times and good-quality results. The FM algorithm, starting from an initial bipartition, performs a number of passes until it finds a locally optimal partition, where each pass consists of a sequence of vertex moves. The fundamental idea is the notion of *gain*, which is the decrease in the cost of a bipartition by moving a vertex to the other part. The local search strategy adopted in the KLFM approach repeatedly moves the vertex with the maximum gain, even if that gain is negative, and records the best bipartition encountered during a pass. Allowing tentative moves with negative gains brings "hill-climbing ability" to the approach.

The K-way HP problem is usually solved by recursive bisection. In this scheme, first, a two-way partition of $\mathcal{H}$ is obtained and then this bipartition is further partitioned in a recursive manner. After $\lg_2 K$ levels, $\mathcal{H}$ is partitioned into K parts. There are also algorithms that try to compute a K-way partitioning directly instead of recursive bipartitioning. The most notable of them is Sanchis's algorithm [29], which is a generalization of FM paradigm to K-way partitioning. In this work, we propose a two-phase approach for K-way declustering. In the first phase, we perform recursive bipartitioning to obtain an initial K-way partition. In the second phase, this initial K-way partition/declustering is improved through a direct K-way refinement heuristic.

The basic idea behind our declustering algorithms is same for database systems with homogeneous and heterogeneous data-item sizes. However, the concepts and the algorithm are simpler to present for homogeneous data-item sizes. So, in the following section, we describe our declustering algorithm for database systems with homogeneous data-item sizes into detail. Then, we briefly summarize the extension of the algorithm to handle heterogeneous data-item sizes in a separate section. Due to the natural correspondence between a database system and a hypergraph, we describe our algorithms using the database-specific notation of Section 2 instead of hypergraph-specific notation, as much as possible, for clarity of presentation.

4.1. Database systems with homogeneous data-item sizes

In this section, without loss of generality, we assume unit retrieval times for data items for simplicity of discussion.

4.1.1. Recursive bipartitioning phase

The objective in recursive bipartitioning phase is to attain a "good" initial K-way declustering for multi-way refinement to be performed in the second phase. Here, we consider a good initial declustering for K-way refinement as even distribution of every query across disks. This even query distribution is assumed to avoid a bad locally optimal declustering providing flexibility in the search space of the multi-way refinement scheme. We have also developed and experimented a more complicated scheme which models, as much as possible, the minimization of the objective function for the final K-way declustering during recursive bipartitioning. In this alternative scheme, we keep track of individual query distribution results obtained in the earlier bipartitioning levels and then use this information to dynamically update the best attainable final response time for each query to relax the objective function for later bipartitioning levels. In the experiments, we observed that although this alternative scheme produces K-way initial declusterings with less aggregate parallel response time than the simpler even query distribution scheme, it leads to worse multi-way refinement results [30].

If K is a power of two, even query distribution objective in the final K-way declustering can be achieved by obtaining an even query distribution at each bipartitioning step through adopting the query splitting scheme shown in Fig. 3. Every bipartitioning step resulting in a bipartition $\Pi_2 = \{\mathcal{D}_0, \mathcal{D}_1\}$ generates two database sub-systems $(\mathcal{D}_0, \mathcal{Q}_0)$ and $(\mathcal{D}_1, \mathcal{Q}_1)$. Each query $q \in \mathcal{Q}$ is split into two item-wise disjoint sub-queries $q' = q \cap \mathcal{D}_0$ and $q'' = q \cap \mathcal{D}_1$. Then, these two queries are added to the query sets $\mathcal{Q}_0$ and $\mathcal{Q}_1$ if $|q'| > 1$ and $|q''| > 1$, respectively. So, the objective of even distribution of these sub-queries at each recursive bipartitioning step models the objective of even distribution of queries into K disks. This scheme can be enhanced to handle any arbitrary K value, which is not restricted to be a power of two, by enforcing properly imbalanced query distributions rather than even distributions in some bipartitioning steps. However, we will only discuss recursive bipartitioning for the case of K being a power of two for the sake of clarity of presentation.

The cost of a bipartition Π_2 according to the "goodness" definition discussed above is

$$cost(\Pi_2) = \sum_{q \in \mathcal{Q}} f(q)\left(\max\{t_0(q), t_1(q)\} - \left\lceil \frac{|q|}{2} \right\rceil \right). \tag{2}$$

As all data items are assumed to have unit retrieval time, $t_k(q)$ denotes the number of data items in part $\mathcal{D}_k$ that qualify for q, i.e., $t_k(q) = |q \cap \mathcal{D}_k|$ for $k = 0, 1$. So, without loss of generality, the gain of moving a data item d from $\mathcal{D}_0$ to $\mathcal{D}_1$ will be

$$g(d) = \sum_{q: d \in q} f(q)(\max\{t_0(q), t_1(q)\} - \max\{t_0(q) - 1, t_1(q) + 1\}). \tag{3}$$

We will restrict our discussion to the contribution of a specific query q that contains d to $g(d)$. Consider the case $t_0(q) \leqslant t_1(q)$, which means that $\max\{t_0(q), t_1(q)\} = t_1(q)$, prior to the move. Since moving d to $\mathcal{D}_1$ will increase $t_1(q)$ by 1, it will increase the cost due to q by $f(q)$, thus q contributes to $g(d)$ by $f(q)$. It is clear from Eq. (3) that the move of d to $\mathcal{D}_1$ will incur a decrease in the cost due to q only if $t_0(q) - 1 \geqslant t_1(q) + 1$. So, query q contributes to $g(d)$ by $f(q)$ if $t_0(q) \geqslant t_1(q) + 2$ prior to the move. Fig. 4 displays our gain-computation algorithm in pseudo-code.

The efficiency of FM-based algorithms depends on the simplicity and efficiency of maintaining

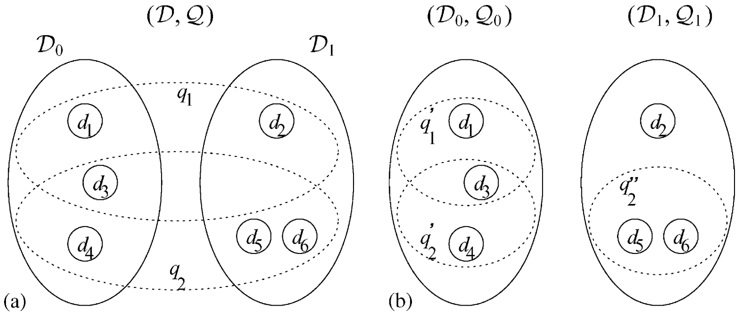


Fig. 3. Query splitting process in recursive bipartitioning.

```
InitializeGains($(\mathcal{D}, \mathcal{Q}), \Pi_2 = \{\mathcal{D}_0, \mathcal{D}_1\}$)
  ▷ Computation of $t_0(q)$ and $t_1(q)$
  for each query $q \in \mathcal{Q}$ do
    $t_0(q) \leftarrow t_1(q) \leftarrow 0$
    for each data item $d \in q$ do
      $k \leftarrow part(d)$    ▷ $part(d)$: index of the part that contains $d$
      $t_k(q) \leftarrow t_k(q) + 1$
  ▷ Computation of move gains for all data items
  for each data item $d \in \mathcal{D}$ do
    $g(d) \leftarrow 0$
    $k \leftarrow part(d)$
    for each query $q$ that contains $d$ do
      $\Delta \leftarrow t_k(q) - t_{1-k}(q)$
      if $\Delta \geq 2$ then
        $g(d) \leftarrow g(d) + f(q)$
      elseif $\Delta \leq 0$ then
        $g(d) \leftarrow g(d) - f(q)$
```

Fig. 4. Initial move-gain computation algorithm for bipartitioning.

move gains through local updates. We propose an efficient local move-gain update scheme as shown in Fig. 5. Our local update scheme is based on the following observation. When a data item d^* with maximum gain is selected to move during the course of the algorithm, only the distributions of the queries that contain d^* change. So, it is sufficient to consider updating the move gains of only the data items that qualify for these queries. Such a query q incurs a gain update only if the move causes a state transition in the distribution of q, where those states are clearly shown in the gain computation algorithm given in Fig. 4. As these states are defined by the difference (i.e., Δ) between the number of data items of q in the source and destination parts and the state transitions are at $\Delta = 2$ and 0 as shown in Fig. 4, the cases to be considered for updating are restricted to those with $-1 \leqslant \Delta \leqslant 3$ prior to the move as shown in Fig. 5.

The overall algorithm can be summarized as follows. The algorithm starts from a randomly constructed initial feasible bipartition. The initial move gains are computed using the algorithm shown in Fig. 4. At the beginning of each pass, all data items are *unlocked*. At each step in a pass, an unlocked data item with maximum move gain (even if it is negative), which does not violate the feasibility criterion, is selected to move to the other part and then it is locked. This locking mechanism, which enforces each data item to be moved at most once during a pass, is needed to avoid thrashing. After the move, the move gains of the affected data items are updated using the algorithm given in Fig. 5. The refinement process within a pass terminates when either no feasible move remains or the sequence of last x moves does not yield a decrease in the bipartitioning cost. This scheme corresponds to realizing a prefix subsequence of feasible moves, which incurs the maximum decrease in the cost. Here, x is the window size that determines the hill-climbing ability. A high x increases the chance of discovering a local minimum that is hidden behind a local maximum at the cost of increasing the running time of the algorithm. Window size is usually selected to be a small fraction of the total number of possible moves (i.e., number of data items) for run-time efficiency. $x = 0.05|\mathcal{D}|$ is used in this work. If the pass terminates due to window-size restriction,

```
UpdateGains($(\mathcal{D}, \mathcal{Q})$, $\Pi_2 = \{\mathcal{D}_0, \mathcal{D}_1\}$, $d^* \in \mathcal{D}_s$)
    for each query $q$ that contains $d^*$ do
        $\Delta \leftarrow t_s(q) - t_z(q)$
        for each data item $d \in q$ do
            if $d \in \mathcal{D}_s$ then
                if $\Delta = 3$ then
                    $g(d) \leftarrow g(d) - f(q)$
                elseif $\Delta = 2$ then
                    $g(d) \leftarrow g(d) - 2f(q)$
                elseif $\Delta = 1$ then
                    $g(d) \leftarrow g(d) - f(q)$
            elseif $d \in \mathcal{D}_z$ then
                if $\Delta = 1$ then
                    $g(d) \leftarrow g(d) + f(q)$
                elseif $\Delta = 0$ then
                    $g(d) \leftarrow g(d) + 2f(q)$
                elseif $\Delta = -1$ then
                    $g(d) \leftarrow g(d) + f(q)$
        $t_s(q) \leftarrow t_s(q) - 1$
        $t_z(q) \leftarrow t_z(q) + 1$
    $part(d^*) \leftarrow z$ (i.e., move $d^*$ to part $\mathcal{D}_z$) and lock $d^*$
```

Fig. 5. Move-gain update algorithm for bipartitioning when d^* with maximum gain is selected to move from source part $\mathcal{D}_s$ to destination part $\mathcal{D}_z$ where $z = 1 - s$.

the last x moves are undone since they do not decrease the cost (they might have actually increased the cost). The initial gain computations for the following pass is achieved through this rollback operation. The overall refinement process terminates if the total gain of a pass is not positive.

Selection of moves with maximum gain necessitates maintaining a priority queue, implemented as a binary max-heap in this work. The priority queue should support extract-max, increase-key and decrease-key operations. Increase-key and decrease-key operations are needed because of the gain increment and decrement operations performed during the gain update computations shown in Fig. 5.

4.1.2. Multiway refinement phase

Although each data item is associated with a single move in bipartitioning, $K - 1$ moves are associated with a data item in multi-way refinement of a K-way declustering. Recall that the cost of a K-way declustering of database system $(\mathcal{D}, \mathcal{Q})$ with homogeneous data-item sizes is

$$cost(\Pi_K) = R_O(\mathcal{Q}) = \sum_{q \in \mathcal{Q}} f(q)(r(q) - r_{opt}(q)), \qquad (4)$$

where $r(q) = \max_{1 \leqslant k \leqslant K} \{t_k(q)\}$ and $r_{opt}(q) = \lceil |q|/K \rceil$. As seen in Eq. (4), a single data-item move may decrease the response time $r(q)$ of a query q that has a non-optimal parallel response time only if there exists only one bottleneck disk D_b for q. In this situation, we say that query q is *critical* to disk D_b since moving a data item that qualifies for q and resides on D_b to another disk D_z may reduce $r(q)$ by one. However, such a positive move gain of $f(q)$ is offset if $t_z(q) = r(q) - 1$ prior to the move so that the move will not change $r(q)$. As also seen in Eq. (4), a move can increase the response time $r(q)$ of a query q only if a data item moves to a bottleneck disk of q. Thus, a query q

contributes a negative gain of $f(q)$ to the moves of its qualifying data items to its bottleneck disk(s).

The FM paradigm is quite suitable and efficient for refining a bipartition. However, the refinement of a K-way partition is much more difficult and complicated than that of a bipartition. A direct generalization of FM paradigm to K-way refinement proposed by Sanchis [29] for hypergraph partitioning is substantially more expensive. The increase in the computational cost is mainly due to the large number of move-gain updates incurring priority queue updates, which is approximately K times greater than that of bipartitioning. There are $K-1$ moves (move directions) associated with each vertex and K parts in the partition, so there are a total of $K(K-1)$ priority queues. Such schemes are stated to be practical for only small K values (e.g., $K<8$).

In this work, we propose an efficient greedy approach to decrease the number of gain update operations by maintaining a single gain value for each data item rather than $K-1$ move gains. The proposed scheme has the nice property of necessitating only one priority queue rather than $K(K-1)$ priority queues. A vertex move can be viewed as a two-stage process: vertex leaves the source disk on which it resides and then arrives at the destination disk. So, the move gain can be considered as the leave gain minus the arrival loss. These two components of a move gain can easily be extracted from the discussion given above. For example, the leave gain of a data item d from disk D_s is equal to the sum of the frequencies of queries that contain d and are critical to D_s. So our basic idea is to select the data items according to their leave gains and after each selection try to realize the best move associated with the selected data item. Note that finding the best move corresponds to finding a destination part that minimizes the total arrival loss for the selected data item. In this work, rather than using the actual leave gains we introduce and use a *virtual leave-gain* concept to associate with data items so that data items are maintained in the priority queue according to these key values. The reasons behind this choice are both declustering quality and run-time efficiency of gain-update operations as will become clear throughout the discussions.

The virtual leave gain $\tilde{g}(d)$ of a data item d that resides on disk D_s is defined as

$$\tilde{g}(d) = \sum_{q \in \mathcal{Q}_+(d,s)} f(q), \text{ where}$$
$$\mathcal{Q}_+(d,s) \equiv \{q \in \mathcal{Q} : d \in q \text{ and } t_s(q) > r_{opt}(q)\}. \qquad (5)$$

That is, each query q that contains d contributes $f(q)$ to $\tilde{g}(d)$ if the number of data items that qualify for q and reside on D_s is greater than the optimal response time of q. This means that it is possible to improve the distribution of query q through moving data item d to an appropriate destination disk D_z. Thus, virtual leave gain $\tilde{g}(d)$ is an upper bound on the actual leave gain.

Consider a sample query q of size 10 in a 4-way declustering with $r_{opt}(q) = \lceil 10/4 \rceil = 3$. For both 4-way distributions [4:3:2:1] and [4:4:1:1] of q, q contributes a virtual leave gain of $f(q)$ to its 4 qualifying data items residing on disk D_1, because $t_1(q) = 4 > 3 = r_{opt}(q)$ in both distributions. Since q is critical to D_1 in the former distribution, this virtual leave gain is equal to the actual leave gain and it corresponds to the possibility of attaining actual move gain of $f(q)$ which can be realized by moving a data item of q from D_1 to D_3 or D_4. However, in the latter distribution, this virtual leave gain only corresponds to an improvement on the distribution of q by a similar move which will make q critical to D_2, thus providing the possibility of attaining optimal response time for q by further moves from D_2. This can be considered as a look-ahead capability in move-gain computations.

The overall algorithm for multi-way refinement can be summarized as follows. The algorithm starts from the initial K-way declustering obtained through recursive bipartitioning as described in Section 4.1.1. The initial virtual leave-gain for every data item is computed using the algorithm shown in Fig. 6. This algorithm also contains the other necessary initialization operations. At the beginning of each pass, all data items are *unlocked.* At each step in a pass, an unlocked data item d^* with maximum virtual leave gain is selected. The $K-1$ actual move gains associated with d^* are computed as shown in Fig. 7. Then, the best move associated with d^*, which does not violate the feasibility constraint, is realized if the respective

```
InitVirtualLeaveGains($(\mathcal{D}, \mathcal{Q})$, $\Pi_K = \{\mathcal{D}_1, \ldots, \mathcal{D}_K\}$)
  for each query $q \in \mathcal{Q}$ do
    $\triangleright$ Computation of $t_{1\ldots K}(q)$
    $t_{1\ldots K}(q) \leftarrow 0$
    for each data item $d \in q$ do
      $k \leftarrow disk(d)$    $\triangleright$ $disk(d)$: index of the disk that contains $d$
      $t_k(q) \leftarrow t_k(q) + 1$
    $\triangleright$ Computation of query response times and related variables
    $r_{opt}(q) \leftarrow \left\lceil \frac{|q|}{K} \right\rceil$
    $r(q) \leftarrow \max_{1 \le k \le K}\{t_k(q)\}$
    $nb(q) \leftarrow |\{D_k : \ t_k(q) = r(q)\}|$
  $\triangleright$ Computation of virtual leave gains
  for each data item $d \in \mathcal{D}$ do
    $k \leftarrow disk(d)$
    $\tilde{g}(d) \leftarrow 0$
    for each query $q$ that contains $d$ do
      if $t_k(q) > r_{opt}(q)$ then
        $\tilde{g}(d) \leftarrow \tilde{g}(d) + f(q)$
```

Fig. 6. Initial virtual leave-gain computation algorithm for multi-way refinement. Note that $nb(q)$, which denotes the number of bottleneck disks for query q, is maintained for each query to simplify the process for testing a query being critical to a disk in move-gain computations as shown in Fig. 7.

```
ComputeMoveGains($(\mathcal{D}, \mathcal{Q})$, $\Pi_K = \{\mathcal{D}_1, \ldots, \mathcal{D}_K\}$, $d^* \in D_s$)
  for each disk $D_k$, $k = 1$ to $K$ $\ni$ $k \neq s$ do
    $g(d^*, k) \leftarrow 0$
    for each query $q$ that contains $d^*$ do
      $\triangleright$ if $q$ is critical to $D_s$
      if $t_s(q) = r(q)$ and $r(q) > r_{opt}(q)$ and $nb(q) = 1$ then
        if $t_k(q) < r(q) - 1$ then
          $g(d^*, k) \leftarrow g(d^*, k) + f(q)$
      elseif $t_k(q) = r(q)$ then    $\triangleright$ if $D_k$ is a bottleneck disk for $q$
        $g(d^*, k) \leftarrow g(d^*, k) - f(q)$
```

Fig. 7. Algorithm for computing the $K - 1$ actual move gains for selecting the best move associated with data item d^* that has maximum virtual leave gain.

gain is positive and then d^* is locked. If the best feasible move has zero gain, then it is realized only if it leads to a better declustering in terms of feasibility.

After a move is realized, the virtual leave gains of affected unlocked data items are updated using the algorithm given in Fig. 8. As expected, possible virtual leave-gain updates are restricted only to the data items that qualify for the queries containing the moved data item. As a fortunate property of virtual leave-gain concept, it is sufficient to consider the update of the virtual leave gains of only those data items that reside on the source and destination disks of the move. This is because of the simple fact that the virtual leave gain of a data item d depends on only the data items that reside on the same disk with d. Comparison of update algorithms given in Fig. 5 and Fig. 8 shows that the virtual leave-gain update algorithm is at least as simple and efficient as the move-gain update

```
UpdateVirtualLeaveGains($(\mathcal{D}, \mathcal{Q})$, $\Pi_K = \{\mathcal{D}_1, \ldots, \mathcal{D}_K\}$, $d^* \in D_s$, $D_z$)
    move and lock $d^*$ to $D_z$
    for each query $q$ that contains $d^*$ do
        $t_s(q) \leftarrow t_s(q) - 1$
        $t_z(q) \leftarrow t_z(q) + 1$
        ▷ Update of virtual leave gains
        for each unlocked data item $d \in q$ do
            if $|t_s(q)| = r_{opt}(q)$ and $d \in D_s$ then
                $\tilde{g}(d) \leftarrow \tilde{g}(d) - f(q)$
            if $|t_z(q)| = r_{opt}(q) + 1$ and $d \in D_z$ then
                $\tilde{g}(d) \leftarrow \tilde{g}(d) + f(q)$
        ▷ Update of response time and number of bottleneck disks
        if $t_z(q) > r(q)$ then    ▷ $r(q)$ increased by the move
            $r(q) \leftarrow t_z(q)$
            $nb(q) \leftarrow 1$
        elseif $t_z(q) = r(q)$ and $t_s(q) < r(q) - 1$ then
            ▷ $r(q)$ not changed by the move
            $nb(q) \leftarrow nb(q) + 1$
        elseif $t_z(q) < r(q)$ and $t_s(q) = r(q) - 1$ then
            if $nb(q) = 1$ then    ▷ $r(q)$ decreased by the move
                $r(q) \leftarrow \max_{1 \le k \le K}\{t_k(q)\}$
                $nb(q) \leftarrow |\{D_k : t_k(q) = r(q)\}|$
            else    ▷ there are other bottleneck disks
                $nb(q) \leftarrow nb(q) - 1$
```

Fig. 8. Virtual leave-gain update algorithm for multi-way refinement when data item d^* is selected to move from disk D_s to D_z.

algorithm for bipartitioning. The multi-way refinement process within a pass terminates when all data items are explored or the last $x = 0.05|\mathcal{D}|$ steps do not lead to a move. The overall refinement process terminates if no move is realized in a pass. Note that the hill-climbing capability of the KLFM paradigm is omitted in our algorithm, because data items are moved only if they lead to non-negative gains.

4.2. Database systems with heterogeneous data-item sizes

In this section, we mainly discuss the extensions to the algorithms presented in Section 4.1 that are necessary to handle data items with different retrieval times. The omitted details should be assumed to be the same or trivially extendible.

4.2.1. Recursive bipartitioning phase

In the case of heterogeneous data-item sizes, the "goodness" of an initial K-way declustering for multi-way refinement corresponds to query distribution with small variance as much as possible. This objective is approximated by balancing the distribution of every query in each bipartitioning step. Thus, for the recursive bipartitioning phase, the cost of a bipartition is

$$cost(\Pi_2) = \sum_{q \in \mathcal{Q}} f(q)(\max\{t_0(q), t_1(q)\} - r_{opt}(q)). \tag{6}$$

Here, $r_{opt}(q)$ is the optimal response time of query q for a 2-way declustering, which is NP-hard to find. Fortunately, it is not necessary to compute $r_{opt}(q)$ for move gain computations since it is a constant. The move gain of a data item d depends on its retrieval time $t(d)$:

$$g(d) = \sum_{q:d\in q} f(q)(\max\{t_0(q), t_1(q)\} - \max\{t_0(q) - t(d), t_1(q) + t(d)\}). \qquad (7)$$

assuming $d \in \mathcal{D}_0$ without loss of generality. Therefore, the move gain of d can be easily computed by comparing $\Delta = t_0(q) - t_1(q)$ with $t(d)$ for every query q containing d. Only three cases need to be checked for initial gain computations as shown in Table 1.

For local gain update, a query q that contains the moved data item d^* incurs a gain update for a data item $d \in q$ only if the move causes a state transition in the distribution of q, where the state transitions are at $\Delta = 2t(d)$ and $\Delta = 0$. The six distinct cases that need to be examined for gain update of data item d that resides in the same part with d^* prior to the move are displayed in Table 2. As seen in the table, the move necessitates a gain update only if $0 < \Delta < 2(t(d) + t(d^*))$ prior to the move. The six cases that need to be examined for updating the gain of a data item that resides in the other part are symmetric and can be derived easily.

Table 1
Initial gain computation for a data item $d \in \mathcal{D}_k$: contribution of a query q that contains d to $g(d)$, where $\Delta = t_k(q) - t_{1-k}(q)$

Case	Contribution of q
$\Delta \geqslant 2t(d)$	$t(d)$
$0 \leqslant \Delta < 2t(d)$	$\Delta - t(d)$
$\Delta < 0$	$-t(d)$

4.2.2. Multi-way refinement phase

For the multi-way refinement phase of heterogeneous case, the virtual leave-gain concept can be generalized as follows for a data item $d \in D_s$:

$$\tilde{g}(d) = \sum_{q:d\in q} f(q)\min\{t(d), \max\{0, t_s(q) - r_{opt}(q)\}\}. \qquad (8)$$

That is, the contribution of a query q that contains d to $\tilde{g}(d)$ corresponds to how much the total retrieval time of data items that qualify for q and reside on D_s approaches from above to the optimal response time of q with the leave of d from D_s. Note that $\tilde{g}(d)$ is an upper bound on the gain of the best move associated with data item d as in the homogeneous case. As the number partitioning problem is NP-hard, $r_{opt}(q)$ values needed for the computation of virtual leave gains are estimated by adapting the best-fit-decreasing heuristic used in solving the K-feasible bin-packing problem [31]. In this heuristic, data items that qualify for a query q are assigned to K bins in decreasing retrieval-time order, where best-fit criterion corresponds to assigning a data item to the bin with minimum sum of retrieval times.

For the local update of virtual leave gains after the move of a data item $d^* \in q$ from disk D_s to D_z, it is only necessary to compute the difference between the values of $\min\{t(d), \max\{0, t_k(q) - r_{opt}(q)\}\}$ for $k = s$ (before the move) and $k = z$ (after the move), to update the virtual move gain of a data item $d \in q$ residing on D_s or D_z.

For the computation of $K - 1$ actual move gains of a selected data item d^*, we maintain a variable

Table 2
Gain update for a data item $d \in \mathcal{D}_s$ when data item $d^* \in \mathcal{D}_s$ is moved to $\mathcal{D}_{1-s}$: change in the contribution of a query q that contains both d and d^* to $g(d)$, where $\Delta = t_s(q) - t_{1-s}(q)$ prior to the move

Case		Contribution of q		
Before move	After move	Before move	After move	Change in contribution
$\Delta \geqslant 2t(d)$	$\Delta - 2t(d^*) \geqslant 2t(d)$	$t(d)$	$t(d)$	0
$\Delta \geqslant 2t(d)$	$0 \leqslant \Delta - 2t(d^*) < 2t(d)$	$t(d)$	$\Delta - 2t(d^*) - t(d)$	$\Delta - 2t(d^*) - 2t(d)$
$\Delta \geqslant 2t(d)$	$\Delta - 2t(d^*) < 0$	$t(d)$	$-t(d)$	$-2t(d)$
$0 \leqslant \Delta < 2t(d)$	$0 \leqslant \Delta - 2t(d^*) < 2t(d)$	$\Delta - t(d)$	$\Delta - 2t(d^*) - t(d)$	$-2t(d^*)$
$0 \leqslant \Delta < 2t(d)$	$\Delta - 2t(d^*) < 0$	$\Delta - t(d)$	$-t(d)$	$-\Delta$
$\Delta < 0$	$\Delta - 2t(d^*) < 0$	$-t(d)$	$-t(d)$	0

$r_2(q) = \max_{1 \leqslant k \neq b \leqslant K} t_k(q)$ for each query q, where b is the index of the bottleneck disk of q (i.e., $r(q) = t_b(q)$). $r_2(q)$ may also be described as the total retrieval time of q in its second bottleneck disk. Keeping track of $r_2(q)$, we can easily decide if a disk is the only bottleneck disk for a query, so we can compute the contribution of a query q to the actual gain of moving $d^* \in q$ from D_s to D_k in constant time as follows:

```
if $t_s(q) = r(q)$ then ▷ if $q$ is critical to $D_s$
    $g(d^*, k) \leftarrow g(d^*, k) +$
    $f(q)(r(q) - \max t_s(q) - t(d), t_z(q) + t(d), r_2(q))$
else
    $g(d^*, k) \leftarrow g(d^*, k) +$
    $f(q)(r(q) - \max r(q), t_z(q) + t(d))$
```

Note that $r_2(q)$ corresponds to $nb(q)$ of the homogeneous case, but $nb(q)$ is more informative taking advantage of the fact that a move can only cause a unit change in the response time of a query.

4.3. Running-time analysis

In the recursive bipartitioning phase, initial-gain computations shown in Fig. 4 take $\Theta(\sum_{q \in Q}|q|)$ time for each FM pass. Each FM pass requires at most $|\mathcal{D}|$ extract-max operations since there can be at most $|\mathcal{D}|$ moves in a pass. Gain-update computations dominate the time complexity of each FM pass. As seen in Fig. 5, the number of gain updates associated with a query q after the move of data item $d^* \in q$ is at most the number of data items in q that are unlocked at the time of the move. Since a data item is locked immediately after being moved, the number of unlocked data items in q becomes one less after each move. Thus, in each FM pass, the number of gain updates associated with a query q is at most $|q|(|q| - 1)/2$. So, gain-update computations necessitate $O(\sum_{q \in Q} |q|^2)$ increase-key and decrease-key operations in a pass. With a binary heap implementation of priority queue, the cost of an FM pass is $O(|\mathcal{D}|\lg|\mathcal{D}| + \sum_q|q|^2\lg|\mathcal{D}|) = O(\sum_q|q|^2\lg|\mathcal{D}|)$. In practice, small number of FM passes ($\leqslant 5$) are sufficient for convergence. So, the computational cost of a recursive bipartitioning step is $O(\sum_q|q|^2\lg|\mathcal{D}|)$. Since $\lceil \lg_2 K \rceil$ levels are involved, recursive bipartitioning phase takes $O(\sum_q|q|^2\lg|\mathcal{D}|\lg K)$ time. This is a rather loose bound since it disregards the decrease in the square terms due to the decrease in the query sizes incurred by the query splitting process. For the homogeneous case, in which the objective is even splits of queries, individual square terms effectively reduce by a factor of two after each recursive bipartitioning level. That is, the total number of priority-queue update operations can be written as $O(\sum_{i=0}^{\lg K-1} \sum_q 2^i(|q|/2^i)^2) = 2((K-1)/K)O(\sum_q|q|^2) = O(\sum_q|q|^2)$. So, for practical purposes, the running time reduces to $O(\sum_q|q|^2\lg|\mathcal{D}|)$ for the homogeneous case.

The proposed multi-way refinement scheme maintains a single gain value (virtual leave gain) for every vertex and hence maintains a single priority queue. So, the running-time analysis given for a bipartitioning step also applies to the multi-way refinement phase. There are only two sources of additional cost. The first one is the computation of $K - 1$ actual move gains for selecting the best move associated with the data item that has the maximum virtual leave gain. As seen in Fig. 7, this additional cost is $O(|\mathcal{D}|K)$. The second one is the update of $r(q)$ and $nb(q)$ ($r_2(q)$ in the heterogeneous case), which is performed when $r(q)$ is affected by the move. As seen in Fig. 8, this operation can be performed in $O(K)$ time for each affected query. Since a query q can be affected by at most $|q|$ moves, this additional cost is $O(K\sum_q|q|)$. Hence, the running time of the multi-way refinement phase is $O(K(|\mathcal{D}| + \sum_q|q|) + \sum_q|q|^2\lg|\mathcal{D}|) = al = 1 > O(K\sum_q|q| + \sum_q|q|^2\lg|\mathcal{D}|)$. Thus, the overall running time of the proposed algorithm is $O(K \sum_q |q| + \sum_q |q|^2\lg|\mathcal{D}|\lg K)$, where the $\lg K$ factor disappears in the homogeneous case as discussed above.

It is possible to reduce the effect of the quadratic term in the complexity of our proposed algorithm with a simple engineering approach. Gain-update operations are much cheaper than the associated increase-key and decrease-key operations on the priority queue. After each move, a data item that shares more than one query with the data item being moved can have its gain updated multiple times. However, the modification of priority queue can be performed only once per data item. Based on this observation, it is possible to modify the gain-update algorithm as follows. During each move, we update the gains of data items without modifying the priority queue while maintaining a list of data items that have their gains updated. At the end of the move, we perform increase-key or decrease-key operations for the data items in this list. This improvement reduces the number of priority-queue updates to the number of edges in the similarity graph of the database system.

The running-time analysis of the WSG method [16] is also given here for the sake of performance comparison. The construction of WSG $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ for a given database system $(\mathcal{D}, \mathcal{Q})$ involves the construction of $\Theta(\sum_q |q|^2)$ edges. The contractions of multiple edges connecting the same pairs of vertices require search in the adjacency lists of the respective vertices. So, the average running time of the graph construction phase can be given as $\Theta(\alpha \sum_q |q|^2)$, where $\alpha = |\mathcal{E}|/2|\mathcal{V}|$ is the average vertex degree in the WSG. The global WSG method [16] is a two-phase approach consisting of recursive-bipartitioning followed by pairwise optimization. In the recursive bipartitioning phase, the running time of an FM pass is $O((|\mathcal{V}| + |\mathcal{E}|)\lg|\mathcal{V}|) = O((|\mathcal{D}| + |\mathcal{E}|)\lg|\mathcal{D}|) = O(|\mathcal{E}|\lg|\mathcal{D}|)$ with a binary heap implementation of priority queue. It is also assumed that constant number of FM passes are sufficient for convergence in each bipartitioning step. As the cut edges are removed after each bipartitioning step, it can also be assumed that $\lceil \lg K \rceil$ recursive bipartitioning levels do not increase the asymptotic complexity. In the pairwise optimization phase, the bipartitioning algorithm is used to refine selected part (disk) pairs until no pairwise improvement is possible. Although this method may require $O(K^2)$ pairwise refinement steps, it converges quickly in practice for sufficiently small K as also mentioned in [16]. So, the running time of the partitioning phase can be assumed to be $O(|\mathcal{E}|\lg|\mathcal{D}|)$. Note that the partitioning time heavily depends on $|\mathcal{E}|$, which depends on the similarities among the queries in $\mathcal{Q}$. A high level of similarity means smaller $|\mathcal{E}|$ in the WSG, which means less partitioning time. However, this also means higher construction time due to the increase in the search time during the contraction of multiple edges between vertex pairs.

The running times of both the proposed and WSG algorithms can be improved for the homogeneous case by using gain-bucket list implementation [28] for the priority queue. This implementation enables almost constant-time priority-queue operations when the range of possible gain values is small. This condition holds for the homogeneous case, so that the $\lg|\mathcal{D}|$ factor disappears.

4.4. Dynamic databases

The proposed two-phase algorithm is mainly well suited for static databases. However, the multi-way refinement scheme of the second phase is very suitable to adjust an existing declustering to updates in a dynamic database. These updates include insertion and deletion of data items as well as changes in the available query information. Independent of the type of updates, it is possible to consider the current status of the database as an initial declustering. Therefore, the multi-way refinement algorithm can be used to refine the declustering to adapt to the updated database. This can be performed periodically using the latest query information.

For periodical refinement, the initial declustering is regarded as a $(K+1)$-way declustering as follows. The data items that already exist in the database inherit their current disk allocation in this initial declustering, thus constituting the allocation for the first K disks, $D_1, \ldots, D_K$. The data items that are inserted after the last refinement are temporarily allocated onto a virtual disk D_{K+1}. The data items that are deleted after the last declustering are no longer in the database, so they are not considered any more. With this setting, periodical refinement is performed in two stages. The purpose of the first stage is to induce a K-way declustering from the $(K+1)$-way declustering by allowing moves only from disk D_{K+1} to disks $D_1, \ldots, D_K$. The K moves associated with each data item on D_{K+1} are inserted into the respective priority queues according to their respective arrival-loss values. Note that leave gains are not considered since all data items will eventually leave disk D_{K+1}. For the same reason, the data items that are currently on D_{K+1} are not considered in arrival-loss computations, i.e., query sizes and $r_{opt}(q)$ values are gradually incremented as data items move from D_{K+1}. At each step of the algorithm, a feasible move with minimum arrival-loss value is selected from these K priority queues. After the move of a data-item d, arrival-loss values of the moves associated with the data items that share queries with d are updated accordingly. The first stage ends when disk D_{K+1} becomes empty. Then, the second stage is carried on as a multi-way refinement over the first K disks as described in Sections 4.1.2 and 4.2.2.

The period of refinement should be chosen carefully depending on the frequency of updates in the database. The amount of change in the database between two successive refinement steps should be small enough to justify the “goodness” of the initial declustering. Note that the proposed refinement scheme does not encapsulate the I/O cost associated with the migration of data items after each refinement. Encapsulating both the improvement of declustering quality and data-item migration cost is a further research issue.

5. Experimental results

The proposed direct declustering (DD) algorithm was tested on a collection of database systems obtained by creating synthetic query sets on real data sets. The mapping-function-based declustering schemes perform well for database systems with uniform structure, but their performance degrades for unstructured database systems (e.g., the more the level of page sharing, the worse the performance of CMD in grid files) [16]. Thus, we use unstructured data and query sets and compare the performance of the proposed algorithm with only the WSG model, which can take advantage of existing query information and can handle non-uniform query and data-item sets. A general implementation of the global WSG scheme described in [16] was used in the experiments. Both DD and WSG algorithms were implemented in C language on a Linux platform. All experiments were performed on a PC equipped with a 2 GHz Intel Pentium-IV processor and 500 MB RAM.

Table 3 shows the properties of the nine database systems used in the experiments. The nine data sets used for constructing the database systems can be classified into four groups. The *Face* data set is a collection of gray-scale face images containing 144 images from the MIT image database, 300 images from PEIPA and 400 images from the ORL image database [32–34]. These 844 images are used to construct an image retrieval system using the algorithm described in [24]. In this algorithm, the significant pixels of the images

Table 3
Properties of database systems used in experiments

Class	Data set	$\|\mathcal{D}\|$	$\|\mathcal{Q}\|$	Avg. query size	Avg. vertex degree in WSG
Image	*Face*	844	1024	23.1	301.7
Function Approx.	*HH*	1638	1000	43.3	367.2
	FR	3338	5000	10.0	69.8
GIS (Point)	*Airport*	1176	2500	22.8	86.1
	Place90	3382	6000	17.9	60.1
GIS (Polygon)	*Park*	1022	2000	20.1	51.3
	Ntar	8952	5000	29.2	103.5
	Bea	10674	10000	26.8	124.4
	State	10827	5000	33.5	114.3

are extracted by multi-resolution wavelet analysis and a number of significant pixels are kept as signature for each image. Thus, each pixel location defines a relation among the images (data items) that contain the pixel in their signature files. As a query is a signature (i.e., a set of pixel locations), the set of all possible pixel locations in the images naturally constitutes the query set.

The second group of data sets consists of multi-feature point data used for function-approximation experiments [35]. The *HH* and *FR* data sets contain 22 784 and 40 768 points in 16 and 10 dimensions, respectively. These data sets are indexed into a grid directory with cell size restricted to 16 points as described in [36]. The resulting grid directory contains 1638 data pages (data items) for *HH* and 3338 data pages for *FR*. A set of synthetic rectangular and diagonal queries is generated assuming Gaussian distribution for both query sides and centers for each data set.

Other data sets consist of GIS data collected from the National Transportation Atlas Databases [37]. The *Airport* and *Place90* data sets contain two-dimensional point data. *Airport* contains the public use airports and landing facilities in the US. *Place90* contains place locations from the 1990 Census Master Area reference file. *Airport*, containing 6735 points, is indexed into a grid file of 1176 pages with cell capacity of 8 points. Similarly, *Place90*, containing 23651 points, is indexed into a grid file of 3382 pages. The *Park*, *Ntar*, *State* and *Bea* data sets contain two-dimensional polygon data. The bounding box of every polygon is considered as a data item for these data sets. *Park* contains the national parks, *Ntar* contains the national transportation analysis regions, *Bea* contains the economic areas, and *State* contains the US boundaries with integrated shorelines. A set of synthetic rectangular and diagonal queries are generated for the GIS data sets as for the function-approximation data sets.

As there is significant locality of data items in spatial database systems, the similarities among queries are more regular, i.e., if a pair of queries share some data item, they are likely to share some other neighboring data items. However, as the number of dimensions for multi-dimensional data sets increase, the amount of such locality decreases. For the image data set, the locality is restricted to pixels, so the variation among queries is high for this database system. Thus, these database systems constitute hard declustering instances that cannot be solved effectively by mapping-function-based strategies since these strategies take advantage of locality.

In Table 3, the cardinalities of data-item and query sets are listed for each database system. Information on average query sizes is also provided to be able to observe the effects of query size on the performances of the algorithms. The average vertex degree in the corresponding WSG of each database system is also displayed. As seen in the table, for data sets with less locality such as *Face* and *HH*, the average vertex degree is higher since the queries are highly irregular. For all database systems, all queries are assumed to have equal relative frequencies, that is $f(q) = 1/|\mathcal{Q}|$ for each $q \in |\mathcal{Q}|$.

All nine database systems listed in Table 3 were used in the experiments for homogeneous data-item sizes. In these experiments, unit storage size and retrieval time are assumed for all data items. Only the four database systems containing GIS polygon data sets *Park*, *Ntar*, *Bea*, and *State* were used in the experiments for heterogeneous data-item sizes. In these experiments, item sizes are taken to be equal to the number of edges in the respective polygons. The retrieval times are taken to be proportional to the item sizes. Disks are assumed to be identical so balanced partitioning of data items into disks is aimed in all experiments.

We have tested $K = 4, 8, 16, 32$ way declustering of every database system. For a specific K value, K-way declustering of a database system constitutes a declustering instance. During recursive bipartitioning phase of both WSG and DD algorithms, initial bipartitions are constructed randomly. So, both algorithms are executed 10 times with different random seeds for each declustering instance. The average performance results are displayed in Tables 4–6. The bottom parts of these tables display the geometric means of every performance figure over all database systems for each K.

Two performance metrics, namely *aggregate parallel response time* $R(\mathcal{Q})$ and *aggregate parallel*

Table 4
Performance comparison for database systems with homogeneous data-item sizes

		Aggregate parallel response time			Aggregate parallel response overhead		% storage imbalance	
Data set	*K*	Ideal	WSG	DD	WSG	DD	WSG	DD
Face	4	6.15	6.83	6.81	0.68	0.66	0.4	0.2
	8	3.32	4.15	4.05	0.83	0.73	1.4	1.2
	16	1.92	2.78	2.60	0.86	0.68	3.7	4.4
	32	1.26	2.03	1.79	0.77	0.53	5.8	3.8
HH	4	11.20	11.99	11.99	0.80	0.79	2.6	0.7
	8	5.84	6.82	6.71	0.98	0.87	5.3	0.5
	16	3.18	4.24	3.99	1.05	0.81	6.9	1.0
	32	1.86	2.86	2.60	1.00	0.74	7.8	2.0
FR	4	2.88	3.32	3.27	0.44	0.39	4.2	0.1
	8	1.72	2.20	2.10	0.48	0.38	8.1	0.2
	16	1.18	1.60	1.48	0.42	0.30	7.4	0.5
	32	1.02	1.26	1.17	0.24	0.15	7.5	1.9
Airport	4	6.09	6.48	6.47	0.39	0.38	1.4	1.3
	8	3.30	3.74	3.72	0.44	0.42	3.5	1.6
	16	1.93	2.37	2.28	0.44	0.36	4.9	3.3
	32	1.28	1.65	1.52	0.38	0.24	6.4	5.8
Place90	4	4.85	5.20	5.18	0.34	0.33	1.2	0.2
	8	2.70	3.07	3.01	0.36	0.30	2.5	0.3
	16	1.65	1.97	1.88	0.32	0.23	5.2	0.5
	32	1.16	1.39	1.30	0.23	0.14	6.4	1.0
Park	4	5.38	5.53	5.54	0.14	0.16	1.4	1.3
	8	2.90	3.09	3.09	0.19	0.19	2.4	3.0
	16	1.73	1.89	1.85	0.16	0.11	5.8	4.9
	32	1.25	1.35	1.30	0.10	0.05	6.1	7.1
Ntar	4	7.68	7.95	7.92	0.27	0.24	0.5	0.2
	8	4.04	4.36	4.31	0.32	0.27	1.4	0.2
	16	2.30	2.52	2.45	0.22	0.15	2.8	0.4
	32	1.52	1.64	1.59	0.12	0.07	5.2	0.8
Bea	4	7.07	7.40	7.38	0.33	0.32	0.5	0.2
	8	3.74	4.13	4.10	0.39	0.36	1.2	0.3
	16	2.14	2.43	2.37	0.29	0.23	2.3	0.3
	32	1.45	1.61	1.55	0.16	0.10	4.2	0.5
State	4	8.75	9.08	9.03	0.33	0.28	0.8	0.1
	8	4.58	4.97	4.89	0.39	0.31	1.9	0.2
	16	2.56	2.86	2.77	0.30	0.20	3.2	0.4
	32	1.62	1.83	1.74	0.21	0.12	5.2	0.3
Geometric means over all database systems								
	4	6.27	6.70	6.67	0.37	0.35	1.1	0.3
	8	3.39	3.88	3.81	0.44	0.38	2.5	0.5
	16	1.99	2.43	2.32	0.38	0.28	4.4	1.0
	32	1.36	1.68	1.58	0.27	0.16	6.0	1.6

Table 5
Performance comparison for database systems with heterogeneous data-item sizes

		Aggregate parallel response time			Aggregate parallel response overhead		% storage imbalance	
Data set	K	Ideal	WSG	DD	WSG	DD	WSG	DD
Park	4	358.5	416.6	400.3	58.0	41.8	4.6	3.5
	8	300.8	329.3	317.2	28.5	16.4	7.6	7.3
	16	295.0	303.2	295.9	8.1	0.8	12.3	9.6
	32	295.0	296.6	295.1	1.5	0.0	11.1	10.9
Ntar	4	869.4	1000.4	988.0	131.1	118.7	2.5	1.9
	8	563.8	675.5	661.1	111.7	97.2	5.0	5.4
	16	473.7	513.8	502.7	40.1	29.0	8.7	8.7
	32	461.9	468.4	465.4	6.5	3.5	9.7	9.9
Bea	4	942.5	1092.5	1089.2	150.0	146.7	1.9	1.9
	8	589.7	727.3	715.5	137.6	125.8	4.0	3.4
	16	477.8	539.5	528.9	61.7	51.1	7.8	7.0
	32	459.7	471.8	468.0	12.1	8.4	9.6	9.6
State	4	755.0	866.9	852.9	111.9	97.9	2.9	1.0
	8	520.1	602.4	586.9	82.3	66.8	5.6	3.6
	16	461.4	484.5	475.9	23.1	14.5	9.1	8.7
	32	457.0	459.7	457.2	2.7	0.2	9.7	9.9
Geometric means over all database systems								
	4	686.3	792.6	778.6	106.3	91.8	2.8	1.9
	8	477.6	558.7	544.7	77.5	60.5	5.4	4.7
	16	419.0	449.2	439.9	26.1	11.5	9.3	8.4
	32	411.3	416.6	414.0	4.2	0.6	10.0	10.1

response overhead $R_O(\mathcal{Q})$ defined in Section 2, are used to measure the qualities of the obtained declusterings. In Tables 4 and 5, the ideal response time refers to the aggregate parallel response time of a strictly optimal declustering if it exists. So, it is effectively a lower bound for the optimal response time. Note that ideal response overhead for a declustering is zero by definition. As all queries have equal relative frequencies, aggregate parallel response overhead of a declustering becomes

$$R_O(\mathcal{Q}) = \frac{\sum_{q \in \mathcal{Q}}(r(q) - r_{opt}(q))}{|\mathcal{Q}|}. \quad (9)$$

So, this value effectively refers to the average deviation from optimal parallel response time per query. Thus, especially for the case of homogeneous data-item sizes, aggregate parallel response overhead provides a general measure to compare the performances of the algorithms for different database systems independent of their sizes. For example, as shown in Table 4, the response overhead values for database systems involving data sets *Face* and *HH*, which were declared to be difficult to parallelize, are substantially higher than those for other database systems.

In the experiments for homogeneous data-item sizes, the objective of declustering fits the balanced partitioning objective as both retrieval times and storage sizes for all data items are assumed to be equal. So we preferred to ignore the feasibility constraint mentioned in Definition 2 during the course of both WSG and DD algorithms in these experiments. Substantially small percent storage imbalance values were obtained as displayed in the last two columns of Table 4. The percent storage imbalance value of a given declustering is computed as $100 \times (W_{max} - W_{avg})/W_{avg}$, where W_{max} denotes the load of the most heavily loaded disk and W_{avg} denotes the load of each disk under perfect load balance condition. Note that for homogeneous data-item sizes, $W_{max} = \max_{1 \leqslant k \leqslant K}|\mathcal{D}_k|$ and $W_{avg} = \lceil |\mathcal{D}|/K \rceil$.

Table 6
Average execution times of WSG and DD algorithms in seconds

Data set	K	Homogeneous				Heterogeneous		
		WSG			DD	WSG		DD
		Construction	Partitioning	Total		Partitioning	Total	
Face	4	0.12	0.29	0.41	0.16			
	8		0.35	0.47	0.22			
	16		0.47	0.59	0.26			
	32		0.59	0.71	0.44			
HH	4	0.34	1.06	1.40	0.20			
	8		1.00	1.34	0.33			
	16		1.09	1.43	0.53			
	32		1.29	1.63	0.79			
FR	4	0.29	0.53	0.82	0.44			
	8		0.41	0.70	0.68			
	16		0.53	0.82	1.09			
	32		0.65	0.94	1.31			
Airport	4	0.16	0.18	0.34	0.12			
	8		0.24	0.40	0.18			
	16		0.24	0.40	0.41			
	32		0.24	0.40	0.84			
Place90	4	0.39	0.53	0.92	0.46			
	8		0.53	0.92	1.05			
	16		0.47	0.86	2.25			
	32		0.71	1.10	2.81			
Park	4	0.10	0.12	0.22	0.08	0.06	0.16	0.41
	8		0.12	0.22	0.11	0.06	0.16	0.72
	16		0.12	0.22	0.16	0.06	0.16	1.14
	32		0.12	0.22	0.19	0.12	0.16	1.96
Ntar	4	1.78	1.35	3.13	0.78	2.35	4.13	3.48
	8		1.65	3.43	1.03	2.47	4.25	5.43
	16		2.12	3.90	1.44	2.65	4.43	8.37
	32		2.76	5.54	1.59	3.59	5.37	13.74
Bea	4	3.15	2.82	5.97	1.70	3.24	6.39	6.25
	8		3.35	6.50	2.42	3.12	6.27	9.84
	16		3.88	7.03	3.91	3.29	6.44	15.75
	32		4.71	7.86	5.92	4.43	7.58	24.97
State	4	2.82	1.65	4.47	1.06	2.71	5.53	3.95
	8		2.00	4.82	1.43	2.86	5.68	5.60
	16		2.53	5.35	2.08	3.00	5.82	8.62
	32		3.35	6.17	3.46	4.00	6.82	12.89
Geometric means over all database systems								
	4	0.47	0.61	1.12	0.35	1.05	2.20	2.44
	8		0.66	1.16	0.54	1.07	2.21	3.83
	16		0.77	1.25	0.86	1.12	2.36	6.00
	32		0.91	1.45	1.23	1.66	2.58	9.65

As seen in Table 4, the proposed DD algorithm performs better than the WSG algorithm for all declustering instances except 4-way declustering of *Park*. The performance gap increases with increasing K in favor of our DD algorithm for all database systems. In terms of the mean parallel response overhead values given at the bottom of the table, the DD algorithm produces 5%, 15%, 35%, and 63% better declusterings than the WSG algorithm for $K = 4$, 8, 16, and 32, respectively. This experimental finding can be attributed to the success of our K-way refinement scheme in comparison with the pairwise optimization scheme of the WSG algorithm. As seen in Table 4, although the percent improvement of DD over WSG is substantially large in terms of parallel response overhead, this improvement is smaller in terms of parallel response time. However, the percent improvement in terms of parallel response time is higher for hard declustering instances involving data sets *Face* and *HH* as compared with other declustering instances.

In the experiments for heterogeneous data-item sizes, percent storage imbalance ratio of 10% is enforced in both WSG and DD algorithms. As seen in the last two columns of Table 5, storage imbalance in all declustering instances is below this threshold except in 16- and 32-way declusterings of *Park*. This is due to high variation on data-item sizes and small number of data items in the *Park* data set. As seen in Table 5, storage imbalance values are comparable in the declusterings produced by WSG and DD algorithms. As also seen in the table, the proposed DD algorithm produces better declusterings than the WSG algorithm in all 16 declustering instances. As in the case of homogeneous data-item sizes, the performance gap increases with increasing K in favor of our DD algorithm for all database systems.

Table 6 shows the comparison of execution times of WSG and DD on database systems used in the experiments. The execution time of WSG is decomposed into construction time and partitioning time. In terms of partitioning times in the homogeneous case, while DD is faster than WSG for small number of disks, it becomes slower with increasing number of disks as expected from the running-time analysis given in Section 4.3. However, as seen in the table, the WSG construction time is significant so that the proposed DD algorithm is faster than WSG in total declustering time in 29 out of 36 declustering instances. Moreover, DD remains to be faster than WSG for all K on average. The instances for which WSG runs faster correspond to the declustering of database systems with high level of locality (e.g., *Airport*, *Place90* and *FR*) across large number of disks. On the other hand, DD runs much faster on database systems with lower level of locality (e.g., *Face* and *HH*). Fortunately, this is consistent with the performance gap between the two algorithms, i.e., there is no tradeoff between declustering time and quality. Consequently, we can conclude that optimization-based declustering techniques provide a powerful alternative to mapping-function-based techniques. Although mapping-function-based strategies fit well to structured database systems, the WSG model can be the best choice for unstructured data sets with high locality. On the other hand, DD is a good alternative for database systems with lower level of locality (e.g., high-dimensional data sets or image databases). Moreover, WSG and DD can be effectively used together for initial partitioning and K-way refinement, respectively.

In the case of heterogeneous systems, the execution time of WSG is not affected since the underlying algorithm remains the same. On the other hand, DD is slower on heterogeneous data compared with homogeneous. This is due to the lg$K$ factor that remains in the running time of recursive bipartitioning phase in the heterogeneous case as discussed in Section 4.3. However, this result actually poses a tradeoff between declustering quality and time when we consider the significance of the performance gap between WSG and DD in heterogeneous database systems.

6. Conclusion

In the literature, vast amount of research is devoted to finding appropriate mapping functions to decluster structured data. There exist many techniques that provide reasonable bounds on the aggregate or maximum query-response time for specific data structures. However, the problem can arise in various applications for which the data may not be structured. In such cases, a general tool for declustering is necessary. The only model in literature that provides such generality is the WSG model, which exploits available information on query and data distribution and data sizes with no restriction on the structure of the data or query sets. Our study contributes to this approach by providing a direct model to solve the problem. Although WSG is an elegant model that has reasonable performance for a wide range of instances, the performance of the algorithm degrades as the degree of locality in the data set decreases. Such degradation is caused by the fact that the model cannot capture the individual multi-item relations among data items as demonstrated in this study. As a result, this study provides a general framework for the declustering problem, which is scalable in terms of the degree of locality of the data.

The second phase of the proposed algorithm introduces an original approach to the problem of multi-way data partitioning. The leave-gain concept provides a reasonable approximation to multi-way move gains. This concept can be extended for other partitioning problems with different cost functions to get a faster way of performing multi-way refinement on partitions. An effective hybrid approach can be refining the declustering generated by the WSG model using the proposed multi-way refinement scheme. Another interesting question will be how to generalize the proposed method for handling systems with heterogeneous disks in terms of storage space or I/O speed.

Acknowledgements

The authors would like to thank Bora Uçar, Ümit V. Çatalyürek, and Enis Çetin for their support. We would also like to thank the anonymous referees for their constructive remarks that helped us improve the quality of the paper.

References

[1] P. Ciaccia, Parallel independent grid files based on dynamic declustering method using multiple error correcting codes, Technical Report, University of Bologna, Laboratory for Computer Science, November, 1994.

[2] Y. Zhou, S. Shekhar, M. Coyle, Disc allocation methods for parallelizing grid files, in: Proceedings of the 10th International Conference on Data Engineering, Houston, IX, 14–18 February, 1994, pp. 243–252.

[3] C. Faloutsos, P. Bhagwat, Declustering using fractals, in: Proceedings of the 2nd International Conference on Parallel and Distributed Information Systems, 1993, pp. 18–25.

[4] N. Koudas, C. Faloutsos, I. Kamel, Declustering spatial databases on a multi-computer architecture, in: EDBT Conference, 1996, pp. 592–614.

[5] J. Li, J. Srivastava, D. Rotem, CMD: A multidimensional declustering method for parallel database systems, in: 18th International Conference on Very Large Data Bases, 1992, pp. 3–14.

[6] B. Moon, A. Acharya, J. Saltz, Study of scalable declustering algorithms for parallel grid files, in: Proceedings of the 10th International Parallel Processing Symposium, Honolulu, HI, 1996, pp. 434–440.

[7] B. Moon, J.H. Saltz, Scalability analysis of declustering methods for multidimensional range queries, IEEE Trans. Knowledge Data Eng. 10 (2) (1998) 310–327.

[8] S. Prabhakar, K. Abdel-Ghaffar, D. Agrawal, A.E. Abbadi, Efficient retrieval of multidimensional data sets through parallel I/O, in: Proceedings of the 5th International Conference on High Performance Computing, 1998.

[9] S. Berchtold, C. Böhm, B. Braunmüller, D.A. Keim, H.P. Kriegel, Fast parallel similarity search in multimedia databases, in: SIGMOD 1997, Proceedings ACM SIGMOD International Conference on Management of Data, 1997, pp. 1–12.

[10] L.T. Chen, D. Rotem, Declustering objects for visualization, in: 19th International Conference on Very Large Data Bases, 1993, pp. 85–96.

[11] T.G. Kwon, S. Lee, Load-balanced data placement for variable-rate continuous media retrieval, in: K.C. Nwosu, B.M. Thuraisingham, P.B. Berra (Eds.), Multimedia Database Systems: Design Implementation Strategies, Kluwer Academic Publishers, New York, 1996, pp. 185–207.

[12] H. Pang, B. Jose, M.S. Krishnan, Resource scheduling in a high-performance multimedia server, IEEE Trans. Knowledge and Data Eng. 11 (2) (1999) 303–320.

[13] S. Prabhakar, D. Agrawal, A.E. Abbadi, A. Singh, T. Smith, Browsing and placement of multiresolution images on parallel disks, in: Proceedings of the Fifth Workshop on I/O in Parallel and Distributed Systems, San Jose, CA, 1997, pp. 102–113.

[14] J. Jensch, R. Lüling, N. Sensen, A data layout strategy for parallel web servers, in: 4th International Euro-Par Conference, 1998, pp. 944–952.

[15] P. Ciaccia, P. Tiberio, P. Zezula, Declustering of key-based partitioned signature files, ACM Trans. Database Systems 21 (3) (1996) 295–338.

[16] S. Shekhar, D.R. Liu, Partitioning similarity graphs: a framework for declustering problems, Information Systems 21 (6) (1996) 475–496.

[17] S. Shekhar, S. Ravada, V. Kumar, D. Chubb, G. Turner, Declustering and load balancing methods for parallelizing geographical information systems, IEEE Trans.

on Knowledge and Data Engineering 10 (4) (1998) 652–655.

[18] M.H. Kim, S. Pramanik, Optimal file distribution for partial match retrieval, in: Proceedings of ACM SIGMOD Conference, 1988, pp. 173–182.

[19] H.V. Jagadish, Linear clustering of objects with multiple attributes, in: Proceedings of ACM SIGMOD Conference, 1990, pp. 332–342.

[20] A. Dasdan, C. Aykanat, Two novel multi-way circuit partitioning algorithms using relaxed locking, IEEE Trans. Computer-Aided Des. 16 (2) (1997) 169–177.

[21] T. Lengauer, Combinatorial Algorithms for Integrated Circuit Layout, Wiley, Chichester, UK, 1990.

[22] U.V. Çatalyürek, C. Aykanat, Hypergraph-partitioning based decomposition for parallel sparse-matrix vector multiplication, IEEE Trans. Parallel Distributed Computing 10 (7) (1999) 673–693.

[23] B. Hendrickson, T.G. Kolda, Graph partitioning models for parallel computing, Parallel Computing 26 (12) (2000) 1519–1534.

[24] C.E. Jacobs, A. Finkelstein, D.H. Salesin, Fast multiresolution image querying, in: Computer Graphics Proceedings, Los Angeles, CA, 1995, pp. 277–286.

[25] S. Zhou, M.H. Williams, Data placement in parallel database systems, in: M. Abdelguerfi, K. Wong (Eds.), Parallel Database Techniques, IEEE Computer Science Press, Los Alamitos, CA, 1998, pp. 203–219 (Chapter 10).

[26] R. Karp, Reducibility among combinatorial problems, in: J.E. Miller, J.W. Thatcher (Eds.), Complexity of Computer Computations, Plenum Press, New York, 1972, pp. 85–103.

[27] B.W. Kernighan, S. Lin, An efficient heuristic procedure for partitioning graphs, Bell System Tech. J. 49 (2) (1970) 291–307.

[28] C.M. Fiduccia, R.M. Mattheyses, A linear-time heuristic for improving network partitions, in: Proceedings of the 19th ACM/IEEE Design Automation Conference, 1982, pp. 175–181.

[29] L.A. Sanchis, Multiple-way network partitioning, IEEE Trans. Comput. 38 (1) (1989) 62–81.

[30] M. Koyuturk, Hypergraph based declustering for multi-disk databases, M.S. Thesis, Computer Engineering Department, Bilkent University, September 2000 (Tech. Rep. BU-CE-0017, http://www.cs.bilkent.edu.tr/tech-reports/2000/).

[31] E. Horowitz, S. Sahni, Fundamentals of Computer Algorithms, Computer Science Press, Potomac, MD, 1978.

[32] E.F. Clark, Pilot Europan Image Processing Archive, http://peipa.essex.ac.uk/ipa/pix/faces/manchester/train/

[33] Massachussets Institute of Technology Image Database, ftp://whitechapel.media.mit.edu/pub/images/faceimages.zip.

[34] The ORL Database of Faces, ftp://ftp.uk.research.att.com/pub/data/att_faces.tar.Z, AT&T Laboratories, Cambridge, 1999.

[35] H.A. Güvenir, I. Uysal, Bilkent University Function Approximation Repository, "http://funapp.cs.bilkent.edu.tr/" (2000).

[36] S. Hanan, The Design and Analysis of Spatial Data Structures, Addison-Wesley, Reading, MA, 1990.

[37] Bureau of Transportation Statistics, National transportation atlas databases, CD-ROM, 1999.