Bulletin of the Technical Committee on

Data Engineering

September, 1994 Vol. 17 No. 3 **IEEE Computer Society**

Letters

Special Issue on Data Placement for Parallelism

Notices

Distribution

IEEE Computer Society
1730 Massachusetts Avenue
Washington, D.C. 20036-1903
(202) 371-1012

Letter from the Editor-in-Chief

One of the few things growing faster than processor speed is database size. If database systems are to succeed with the truly large (petabyte) databases of the future, they will need to exploit parallelism. Not only will user queries require the processing of enormous amounts of data, but the users will expect the results to be produced in a timely fashion, i.e. with blinding speed. Applications of database systems to science, to data mining, to multimedia, all will depend on parallelism as a primary technique for achieving acceptable and scalable performance.

Simply scheduling extra processors for some logical partitioning of the data will not suffice. The database system, either automatically or under DBA direction, will need to place the data on disk or other stable media in such a way that parallelism can be effectively exploited. That is what this special issue of the Bulletin is all about. It is a subject of enormous practical importance that generates research interest because the problems are difficult and can be attacked with the analytic skills possessed by strong researchers.

Shahram Ghandeharizadeh, who served as the editor of this issue, has succeeded in bringing together highly informative articles from outstanding researchers in the area of data placement for parallelism. Shahram is himself highly regarded for his work in this area. The results of his editting reflect his strong skills and good judgment on this challenging subject. I want to thank Shahram for this fine job.

The current issue of the Bulletin has a minor stylistic change. The LaTex font previously used for the Bulletin has been replaced by the Times-Roman Postscript font. This has a number of subtle advantages. First, the hardcopy remains readable while being somewhat more dense, hence permitting authors more words per issue. Second, because this is a built-in font, the size of the Bulletin Postscript files is somewhat reduced. Finally, the Postscript viewers that I have do a much better job of rendering the new font, the old LaTex font being nearly unreadable, while the new font can be readily deciphered.

I am happy to be able to report some good news on the state of the Bulletin finances. The TC on Data Engineering has been assured by the IEEE Computer Society that funds will be provided that will allow us to publish the Bulletin in hardcopy for the remainder of this year. (Our previous budget situation did not permit this. Indeed, I was prepared to announce that the current issue would be available only electronically.) For the longer term, the Computer Society has expressed its support for TC activities, including the Bulletin, and will continue to work with us to solve the long term problem of providing hardcopy distribution of the Bulletin.

David Lomet
DEC Cambridge Research Lab
lomet@crl.dec.com

Letter from the Special Issue Editor

The emergence of both the Information Super Highway and the National Information Infrastructure initiatives have added to the increasing interest in parallel information systems. This is because both initiatives envision the use of high performance systems that: (a) provide on-line access to vast amount of data, (b) continue to provide information services in the presence of hardware failures, and (c) scale to thousands of storage/processing elements in order to enable the platform to grow as the requirements of an application grows. Parallel database management systems (DBMS) satisfy these requirements rather nicely.

Using a multi-node[1] hardware platform, a parallel DBMS disperses data across multiple nodes. This minimizes the time required to process a query because the system can partition the query (transparently) into multiple subqueries with each subquery processing a stream of data from a node containing the relevant data. In order to maximize the processing capability of the system, it is essential for the data to be placed across the nodes such that the workload of an application is evenly distributed across the available resources. Otherwise, a single node of a thousand node system may become a bottleneck, reducing the overall processing capability of the system and limiting its scalability characteristics.

The focus of this special issue is on the placement of data in parallel database systems. In the first article, Srivastava et. al., survey techniques to horizontally decluster data in parallel relational DBMS based on a shared-nothing architecture. Next, Golubchik and Muntz provide a tutorial on the role of parity and replication to enable a system to continue operation in the presence of disk failures. This study also describes how the redundant data can be used during the normal operation in order to further enhance the performance of the system. The third article by Schuermann et. al., presents and evaluates a heuristic to dynamically redistribute data in a multi-disk system, enabling the system to respond to the users' evolving pattern of access to the data. In the final article, Tomasic and Garcia-Molina present research issues in parallel document retrieval systems. As a collection, these articles identify some of the issues, solutions, and challenges faced by the researchers investigating the placement of data in parallel information systems.

I would like to take this opportunity to thank the authors for contributing articles to this special issue. I would also like to thank the external reviewers for their time spent reading and providing valuable feedback to the authors, they include: D. Agrawal, A. El Abbadi, H. Hsiao, T. Ieong, V. Krishnaswamy, M. Muralikrishna, D. Schneider, and D. Schrader. Finally, I would like to thank David Lomet, Editor in Chief of the bulletin, for his advice and help with this issue.

Shahram Ghandeharizadeh
Computer Science Department
University of Southern California
Los Angeles, CA 90089

[1] The definition of a node is architecture dependent. While in a multi-disk architecture a node may correspond to a disk drive, in a shared-nothing architecture a node may consist of a CPU, one disk drive and some random access memory.

Data Declustering in PADMA: A PArallel Database MAnager

Jaideep Srivastava, Thomas M. Niccum, Bhaskar Himatsingka
Computer Science Department
University of Minnesota
{srivasta|niccum|himatsin}@cs.umn.edu

1 Introduction

Parallel processing of database operations was first addressed by the database machine community, where the focus was on designing special-purpose hardware [2]. However, the cost of building special-purpose hardware is high, and most of the proposals were never realized [2]. The eighties saw the emergence of very powerful and scalable commercial massively parallel processors (MPPs), with extremely attractive price/performance ratios, e.g. nCUBE, Intel's iPSC and Paragon, KSR-1, and Thinking Machines CM-2 and CM-5. Also, with very high speed communication switches becoming commercially available, e.g. ATM and Fiber Channel, and advancements in operating system technology to make communication cheaper, e.g. Active Messages [27], a network of workstations (NOWs) [20] can be configured to provide the performance and price/performance of scalable parallel machines. Thus, while special-purpose hardware design for databases did not succeed, use of MPPs or NOWs for building parallel databases is an extremely promising and active research area.

The past few years have seen growing activity in the area of parallel databases. The relational data model, whose set-oriented non-procedural nature provides opportunities for massive parallelization, has been found especially suitable [2]. A number of parallel database projects have been started in academia [3, 7] and industry [1, 21] and products such as Parallel Oracle, Tandem Himalaya, Sybase Navigator, Teradata parallel database on NCR DBC/1012, etc. are available in the market. Applications targeted range from transaction processing to deductive databases.

In this paper we provide a brief overview of the ongoing PArallel Database MAnager (PADMA) project at the University of Minnesota and summarize specific results obtained in the area of data declustering. We finally outline the project status and future directions.

2 Data Declustering : An Overview

Record-oriented data can be visualized as points in multi-dimensional space, with the co-ordinate of a point on an axis being the value of the corresponding attribute of the record it represents. The declustering problem thus, is deciding how to partition the entire data space into subspaces, which may be overlapping or non-overlapping, and then deciding how to allocate data subspaces to disks. In general, it is possible to have multiple subspaces allocated to the same disk, as well as a subspace allocated to multiple disks (replication). A data point (record) is stored on the disk(s) to which the subspace containing it is allocated. Several declustering techniques have been proposed in the literature, and good surveys are provided in [6] [15] [12]. A major class is of *single-attribute declustering* methods, where the space partitioning is based on a single attribute. Examples are [9] [2], where the most frequently queried attribute is used for declustering. Another classification of declustering methods can be based on whether the partitioning of the data space is done in terms of regular grids, e.g. *grid-file* [19] type partitioning, or irregular shapes [10]. Though the question of whether regular or irregular partitions are better is by no means settled, our focus has been on finding good declustering methods for regular grid-based partitioning

or *cartesian product files* All these methods and the class of methods we are studying work well for read-only databases. They also handle well behaved updates, though further study is required in this area. Components of the declustering problem are:

- Creating the grid partitions, i.e. dividing the complete data space into regular sub-spaces.
- Assigning individual sub-spaces to disk(s) i.e. *disk allocation*. Since we do not consider replication, this is equivalent to finding a mapping which maps each sub-space to a unique disk.

Various deterministic methods have been studied [17] for creating a grid partitioning of the data space. However, these techniques are applicable only to small data files. Hence, statistical sampling based approaches become extremely important. We have studied sampling based approaches for creating the grid partitions, and the techniques have been shown to have very good partitioning properties. Details of these techniques are provided in [17]. In the following sections, we assume that the grid partitioning has been created. We thus use the terms data declustering and disk allocation interchangeably.

2.1 Problem Definition

We now define some terminology which is used through out this paper. These definitions are similar to those used by Faloutsos et al [5].

Symbol Definitions

Symbol	Definition
M	Number of disks
d	Number of attributes
D_i	Domain of i -th attribute
d_i	Number of Intervals of the domain of the i -th attribute
diskOf()	Function that maps bucket-ids to disks

Definition 1 [Cartesian Product File] Let D_i denote the domain of the i^{th} attribute of a d -attribute file. Let each D_i be partitioned into d_i disjoint intervals $I_{i0}, I_{i1}, ..., I_{id_i-1}$. We call F a *cartesian product file* if all records in partition $I_{1i_1} \times I_{2i_2} \times ... \times I_{di_d}$, where each $I_{ji_j} \in \{I_{j0}, I_{j1}, ..., I_{jd_j-1}\}$, lie in the same unique bucket(disk block). The bucket $b \equiv I_{1i_1} \times I_{2i_2} \times ... \times I_{di_d}$ is denoted by $< i_1, i_2, ..., i_d >$.
Definition 2 [Range Query] A *range query* $Q = [L_1, U_1) \times [L_2, U_2) \times ... \times [L_d, U_d), L_i, U_i \in D_i$, is represented as a d -tuple $([L_1, U_1), [L_2, U_2), ..., [L_d, U_d))$. Here $[L_i, U_i)$ is the range on the i^{th} attribute. Records that satisfy this query must be points that lie in the d -dimensional box $[L_1, U_1) \times [L_2, U_2) \times ... \times [L_d, U_d)$.
Definition 3 [Partial Match Query] A *partial match query* Q is a range query such that
$\{(\exists i)[L_i, U_i) \equiv D_i] \wedge [(\forall j \in \{1, 2, ..., d\})(j \neq i)(L_j = U_j)]\}$.
Definition 4 [Point Query] A *point query* Q is a range query such that
$[(\forall i \in \{1, 2, ..., d\})(L_i = U_i)]$.
Definition 5 [Length of Query] Let $Q = ([L_1, U_1),\ [L_2, L_3), \cdots,\ [L_d, U_d))$ be a range query. The *length* of Q on dimension i is the number of intervals intersecting $[L_i, U_i)$ on dimension i.
Definition 6 [Response Time] The *response time* of a query is defined as : max $(N_0, N_1, ..., N_{M-1})$ where $N_i (0 \leq i \leq M-1)$ is the number of qualifying buckets on disk i, for the query.

Since I/O is the major bottleneck in query processing, it is desirable that I/O be parallelized as far as possible. This becomes particularly important for a query which occurs frequently in a database system. The following

definition of *query optimality* gives the maximum possible I/O parallelization feasible for a query.
Definition 7 [Query Optimality] An allocation method on M disks is *query optimal for a query* Q if the response time of query Q is $\lceil \frac{P}{M} \rceil$, where P is the total number of qualifying buckets for query Q.
Definition 8 [Strict Optimality] An allocation method is *strictly optimal* if it is query optimal for all possible queries. It is *strictly optimal for partial match queries* if it is query optimal for all possible partial match queries. It is *strictly optimal for range queries* if it is query optimal for all possible range queries.

2.2 Survey of Grid Based Declustering Techniques

We now provide a brief overview of the multi-attribute grid-based declustering approaches. These descriptions are only to recapitulate their salient points. Detailed descriptions exist in the respective papers.

Figure 1 provides an example of how each of these techniques allocates a 2 dimensional grid, with 8 intervals on each dimension, onto 4 disks.

1. **Disk Modulo (DM) / Coordinate Modulo Declustering (CMD)** The disk modulo method by Du and Sobolewski [4] and coordinate modulo declustering by Li et al [14] are similar approaches. A bucket $< i_1, i_2, ..., i_k >$ is assigned to the disk unit diskOf$(i_1, i_2, ..., i_k) = (i_1, i_2, ..., i_k) \bmod M$. Variations of this method include the Generalized Disk Modulo allocation method [4].

2. **Field-wise Exclusive-or (FX)** This allocation method was proposed by Kim and Pramanik [16] with efficient partial match retrieval in mind. The main idea behind this approach is the use of bitwise exclusive or operation $(\otimes)$ on the binary values of a bucket-id. If $< i_1, i_2, ..., i_k >$ is a bucket-id then the FX method allocates it to disk unit diskOf$(i_1, i_2, ..., i_k) = T_M[i_1 \otimes i_2 \otimes ... \otimes i_k]$ where T_M is a function which returns the rightmost $log_2 M$ bits of $i_1 \otimes i_2 \otimes ... \otimes i_k$. Since $(\otimes)$ is a boolean operation the values $i_1, i_2, ..., i_k$ must be encoded in binary.

3. **Error Correcting Codes (ECC)** A declustering approach based on using error correcting codes was proposed by Faloutsos et al [6]. It works for binary attributes or an attribute where the number of partitions on it, d_i, is a power of 2. For the binary case the problem is reduced to grouping the 2^k binary strings on k bits in M groups of dissimilar strings. The main idea is to form groups of strings such that each group forms an Error Correcting Code (ECC). In case d_i is a power of 2, the binary representation of the domain is used. Thus if each d_i can be represented as a binary string of length m then we need to construct an ECC on km bits out of which $log_2 M$ bits will be parity check bits while the rest will be information bits.

4. **Hilbert Curve Method (HCAM)** A declustering method based on space filling curves was recently proposed by Faloutsos and Bhagwat [5]. Such a curve visits all points in a k -dimensional grid exactly once and never crosses itself. Thus, it can be used to linearize the points of a grid. The authors use such a curve, called the Hilbert Curve [5] to fill the k -dimensional grid and then assign the disks to the buckets in a round robin fashion. Thus, if H is the function which imposes the linear ordering generated by the Hilbert Curve on the grid points (buckets) then diskOf $(i_1, i_2, ..., i_k) = H(< i_1, i_2, ..., i_k >) mod M$.

2.3 Declustering and Optimality

Ideally, we would like a declustering method to be *strictly optimal*. Queries can vary from being completely specified, partial match, range queries, correlational(diagonal) queries, etc. Consider a cartesian product file F which has $N, N > M$ total grid-blocks. Since the number of grid-blocks are greater than the number of disks, at least two grid-blocks (buckets) will be mapped to the same disk. We can always come up with a query (using union) which accesses exactly these two buckets. Thus, this particular query will not be optimal. This is independent of the declustering method used. Thus, it is not possible to have a declustering method which is *strictly optimal*.

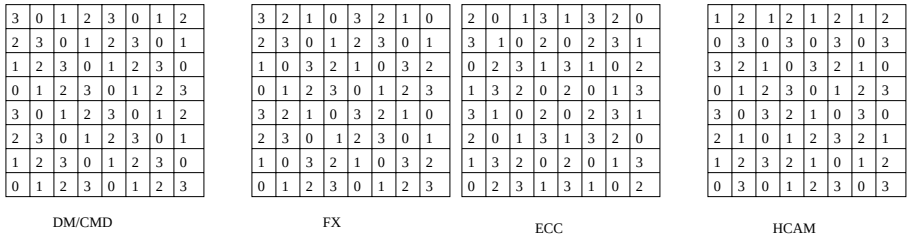


Figure 1: A Declustering Example

From a practical viewpoint, however it can often suffice to consider range and partial match queries only since these are the most commonly occurring class of queries in a database. Given that a declustering method cannot be *strictly optimal*, it is thus desirable to have a declustering method that is *strictly optimal for partial match and range queries*. Much work has been done in proving results about performance bounds of partial match queries for different declustering techniques [4] [6] [16] [8]. Some results also exist about the conditions under which a strictly optimal declustering can be achieved for partial match queries [4] [16] [15].

Recent work [15] has derived sufficient and necessary conditions for optimality of a declustering technique with respect to partial match queries when the number of partitions on all attributes are less than the number of disks. The specific focus of [15] was on p -ary cartesian product files where $(\forall i)(d_i = p)$. It was shown that there is no strictly optimal allocation for a p -ary cartesian product file if

$p^{(p^2+p-2)/2} \leq M \leq p^{d-1}$ or $p^2 \leq M \leq p^{n-p^2-p+2} - 1$.

Thus, for all practical purposes the non existence of strictly optimal declustering methods with respect to partial match queries was shown when the number of partitions on all attributes is less than the number of disks. Since range queries are a superset of partial match queries these results hold for range queries too.

The above result while certainly of theoretical interest is not disheartening from a practical viewpoint. For most medium to large databases having more partitions on an attribute than the number of disks is expected to be quite common. For example, for a 16 disk system, with $8KB$ disk blocks, 64 bytes/record and 3 declustering attributes in a relation, only 1 million records are needed in a relation to have 16 partitions on each dimension, which is not unrealistic for a database requiring parallel processing.

The following discussion shows that range queries place more constraints than partial match queries, and optimality for them is harder to achieve. Specifically, a significant observation is that while the condition $(\forall i)(d_i \geq M)$ guarantees optimality for partial match queries under many conditions, it does not do so for range queries.

Lemma 1. If $M = ab$ is a composite integer and $(\exists i, j)(d_i \geq a + 1, d_j \geq b + 1)$ then a strictly optimal declustering for range queries does not exist.
Proof: Refer [11].

Lemma 2. If M is a prime integer and $(\exists i, j)(d_i \geq 3, d_j \geq M)$ then a strictly optimal declustering for range queries exists iff
$(1) M = 1, 2, 3, 5$ and $d = 2$ or
$(2) M = 1, 2, 3$ and $d \geq 3$.
Proof: Refer [11].

Theorem 1. If $d \geq 3$ then a strictly optimal declustering for range queries exists iff $M = 1, 2, 3$.
Proof: This is a direct consequence of *Lemmas 1 and 2*.

In the following table we summarize the main optimality results for various declustering methods.

Declustering Techniques and Optimality

Declustering Method	Restriction on Number of Disks	Restriction on Number of Partitions	Conditions on Optimal Queries
DM/CMD	None	None	PM :Exactly one field unspecified
			Range/PM: if one of the range domains is an integral multiple of M
FX	power of 2	None	PM: Exactly one field unspecified
		Power of 2	PM: with an unspecified attribute s.t. $d_i \geq M$
ECC	power of 2	power of 2	None derived
HCAM	None	None	None derived

3 Latin Hypercube Declustering Methods (LHDM)

Latin Squares [28] are two-dimensional structures which show very good properties, and have been widely used in experimental designs to ensure least redundancy and maximum coverage for the minimal experimental effort. We generalize Latin Squares into higher dimensions and define a class of declustering methods called *Latin Hypercube Declustering Methods (LHDM).*

Definition 9 [Latin Squares] A *Latin Square* of order n is an $n \times n$ square composed of symbols from 0 to $n-1$ such that no symbol appears more than once in a row or column [28]. Zhou et al discuss some properties of declustering methods using Latin squares in [28].

Definition 10 [Latin Hypercubes] A *Latin Hypercube* of dimension d and order n is an $n \times n \times ... \times n$ hypercube of dimension d composed of symbols from 0 to $n-1$ such that no symbol appears more than once in any row for all dimensions.

Definition 11 [Latin Hypercube Declustering Methods (LHDM)] A declustering method which uses a Latin Hypercube of dimension d and order M as its basic building block is called a *Latin Hypercube Declustering Method*. The hypercubes are replicated along each dimension till they fill up the domain space of the relation. In case the domain space in some attribute is not a multiple of M then the last hypercube in that dimension is incomplete.

In the following discussion we use the term *Latin Hypercube* and *Latin Hypercube Declustering Method* interchangeably. This is not to imply that the complete grid is mapped as a latin hypercube but that it is mapped using a latin hypercube as a basic block. We now derive some basic properties of *Latin Hypercubes* and show sufficient and necessary conditions for a method to belong to the class of *Latin Hypercubes*.

Definition 12 [Periodic Allocation] A declustering method is said to be *periodic* if
$(\forall j \in \{1, 2, ..., d\}) diskOf(< i_i, i_2, ..., i_j, ..., i_d >) = diskOf(< i_i, i_2, ..., i_j + M, ..., i_d >), i_j + M \leq d_j$

Definition 13 [Row Optimal Allocation] A declustering method is said to be *row optimal* if the declustering method is optimal for all queries such that the length of the query is 1 on all but one declustering attribute.

Lemma 3. If a declustering method is *row optimal*, then it is *periodic*.
Proof: Refer [28].

Theorem 2 A declustering method belongs to the class *LHDM* iff it is row optimal.
Proof: Refer [12].

Corollary: DM/CMD, GDM, FX, and Latin Squares belong to class LHDM.
Proof: DM [4], GDM [4], FX [16], and Latin Squares [28], each have been shown to be row optimal in the respective papers. Using Theorem 2, all these methods belong to the class LHDM. **Q.E.D.**

3.1 Performance Analysis of LHDM

In this section we analyze Latin Hypercube Declustering Methods and derive conditions under which optimal parallelism is achieved. To help understand the performance of queries when these conditions do not hold, we also derive upper bounds on the worst case behaviour of all queries. Finally, to understand the expected performance of LHDM we analyze their average case behaviour on queries. All of these results are applicable to any declustering method which belongs to the class LHDM, e.g. CMD, FX, GDM, etc. The proofs to the the Lemmas and Theorems in this section can be found in [12].
Definition 14 [Interval Domain Space] Any query on the cartesian product file F will have to access all the data in the interval it intersects. Thus, the range on any dimension i, of a range query can be transformed to the coordinate system of the interval domain $0 \leq l_i \leq d_i$. We define this grid with the interval domains as its axes as the *Interval Domain Space*.
Definition 15 [Hyper-rectangle] A *Hyper-rectangle* H is a subspace of the d-dimensional interval domain space such that if intervals I_{ik}, I_{il} intersect H on dimension i then $\forall(j)(k \leq j \leq l) I_{ij}$ intersects H. It can be observed that any range query can be represented as a hyper-rectangle in the interval domain space.

Theorem 3. *LHDM* is query optimal for all range queries whose length on some dimension is equal to kM where $k \geq 1$.
Note that Theorem 3 provides only sufficient conditions under which queries are optimal. Thus, it is possible to have queries which do not satisfy this condition and are still optimal. Next we characterize a subset of such queries.

Lemma 4. Let Q be a range query which needs to examine hyper-rectangle $A = \times_{i=1}^{d}(L_i,\ L_i + l_i - 1)$, where $0 \leq L_i \leq d_i - l_i$ and $1 \leq l_i < M$ for $1 \leq i \leq d$. Without loss of generality, let $l_{i_1} \leq l_{i_2} \leq \cdots \leq l_{i_d}$, where $l_{i_k} \in \{l_1,\ ...,\ l_d\}$ for $1 \leq k \leq d$, Q is required to access at most $\prod_{k=1}^{d-1} l_{i_k}$ buckets on each disk.

Lemma 5. Let
$A = \times_{i=1}^{d}(L_i,\ L_i + k_i M + l_i - 1)$,
$A_1 = (L_1,\ L_1 + k_1 M - 1) \times (\times_{i=2}^{d}(L_i,\ L_i + k_i M + l_i - 1))$,
$A_l = (A - \bigcup_{t=1}^{l-1} A_t) \bigcap R_l \quad for\ 2 \leq l \leq d$
where,
$R_l = (\times_{t=1}^{l-1}(L_t,\ L_t + k_t M + l_t - 1)) \times (L_l,\ L_l + k_l M - 1) \times (\times_{t=l+1}^{d}(L_t,\ L_t + k_t M + l_t - 1))$,
$A_{d+1} = \times_{i=1}^{d}(L_i + k_i M,\ L_i + k_i M + l_i - 1)$,
where $0 \leq L_i \leq d_i - k_i M - l_i$, $0 < l_i < M$ for $1 \leq i \leq d$.
A is a hyper-rectangle in S. Thus, all A_i's are hyper-rectangles in S for $1 \leq i \leq d+1$ and have the following properties:
1. $A = \bigcup_{i=1}^{d+1} A_i$.
2. The length of A_i on dimension i is $k_i M$ for $1 \leq i \leq d$.
3. $A_i \bigcap A_t = \emptyset$ if $i \neq t$ for $1 \leq i,\ t \leq d+1$, where $\emptyset$ is empty set.

It is obvious that the hyper-rectangles in F required by any range query, which do not satisfy the condition of Theorem 3, can be represented by A in Lemma 5. Theorem 4 characterizes a subset of such queries for which LHDM is still optimal.

Theorem 4. Let Q be the same range query as A in Lemma 5. *LHDM* are optimal for Q if $(1/B + l_{i_d}/M) > 1$, where $B = \prod_{j=1}^{d-1} l_{i_j}$ and $l_{i_1} \leq l_{i_2} \leq \cdots \leq l_{i_d}$.

Theorem 5. For any range query Q required to examine P buckets, at most

$$\lceil P/M \rceil + (M-1)^{d-1} - 1$$

buckets are accessed per disk in response to Q.

Assumption: For the following discussion we make the assumption $(\forall i)(d_i = nM)$.

Lemma 6. Assume that for any attribute i, all ranges $[L_i, U_i), L_i, U_i \in D_i$, could occur with equal probability in any range query. Now, the probability of any range query being optimal is at least

$$p = 1 - \left(\frac{nM^2 - (n-1)M - 2}{nM^2} \right)^d .$$

Clearly, we can make p large enough by properly selecting n. The probability of a range query not being optimal is less than $1 - p$. The above result shows that the performance of *LHDM* improves with the dimensionality of the data.

Let range query Q be required to examine hyper-rectangle $A = \times_{i=1}^{d} (L_i, L_i + k_i M + l_i - 1)$ containing P buckets, where $0 \leq L_i \leq nM - k_i M - l_i$ and $0 \leq l_i < M$. Assuming l_i's are independently and uniformly distributed in $\{0, 1, ..., M-1\}$, we have the following theorem.

Theorem 6. In response to the range query Q above, at most

$$\lceil P/M \rceil + (1-p) \left(\frac{(M-1)^{d-1}}{2^{d-1}} - 1 \right)$$

buckets are accessed per disk on the average.

Theorems 5 and 6 provide two upper bounds which provide insight into the expected behaviour of LHDM. However, the bounds are not the tightest possible and hence the actual performance of *LHDM* can be much better. Since theoretical analysis was rapidly getting intractable, we decided to carry out an experimental evaluation to study the behaviour of LHDM in more detail. These are described in the next section. One of the most promising applications of parallel databases is in decision support applications running against very large databases. In such scenarios range queries are usually expected to examine a very big subspace of F, i.e. P in Theorems 5 and 6 will be very large. Thus $\lceil P/M \rceil$, the optimal number of disk accesses, is much greater than $(M-1)^{d-1} - 1$ or $(1-p)((M-1)^{d-1}/2^{d-1} - 1)$. And hence, *LHDM* is expected to behave nearly optimally for most range queries.

4 Experimental Evaluation

We believe that while theoretical studies such as [15] [4] [16] [14] and that presented in the previous section, provide valuable insight into the properties of declustering methods, the picture is not complete without a detailed

experimental evaluation. This is more so because of the fact that all declustering techniques are not amenable to detailed theoretical analyses and the bounds obtained are not exact in most cases. Specifically, since in practice no restrictions can be placed on the size and shape of queries, as well as the number of attributes or their domain sizes, we believe an evaluation is needed which varies these dimensions as parameters and studies their effects on the performance of various declustering methods. Thus, we have chosen to carry out experimental evaluations to examine the performance of LHDM. The aim is to see how the different techniques belonging to this class compare amongst themselves and also with other prominent techniques proposed in literature. We choose two declustering methods from the class LHMD, namely **FX** and **CMD**, and two others namely **ECC** and **HCAM**, for our experimental evaluation.

The main results of the experiments [11] are as follows: (i) various declustering methods proposed in literature show a noticeable difference of performance (in relative terms though not much in absolute terms) for small queries, (ii) for large queries, Latin Hypercubes perform very well, (iii) the performance of declustering methods is quite sensitive to the query shape and Latin Hypercubes show better performance for linear queries and (iv) the deviation of most declustering methods from optimality decreases as the number of dimensions, i.e. the number of attributes, of the query box is increased, and specially so for Latin Hypercubes.

Our overall conclusions are that (i) no declustering method can be optimal for all queries on a large database, (ii) for large databases and large queries Latin Hypercube methods perform very well and are not very far from optimal, (iii) information about commonly posed queries can be useful in selecting one method over another much like physical database design in centralized and distributed databases, and this choice is crucial for small queries, and (iv) since different methods may turn out to be better for different relations, based on the queries posed on them, commercial DBMSs will have to support more than one declustering method, much like different kinds of access methods and index structures in today's databases. Based on our studies, future work in the area of declustering must address issues such as (i) how do grid-based methods perform compared to non grid-based ones, (ii) how do various methods perform when data skew and attribute correlation is present, and (iii) how can information about query sets be used in selecting a declustering method appropriate for a relation.

5 Project Architecture, Status & Future Directions

In the last few years, three parallel database (software) architectures have been considered, namely *shared-memory, shared-disk*, and *shared-nothing* [25]. Shared-memory architectures suffer from scalability problems, and thus grew out of favor. An initial consensus was that shared-nothing architectures are the most promising [2, 25], though recently support has been expressed for the shared-disk architecture [26]. We believe that as technology is progressing, the distinction between the latter two is becoming blurred. This is because the shared-disk parallel database architecture is most suited to MPPs, while shared-nothing architecture is most suited to NOWs. However, since MPPs and NOWs are becoming comparable from a hardware viewpoint, i.e. aggregate processor cycles, communication bandwidth and latency, and I/O bandwidth and latency[1] , the distinction between shared-nothing and shared-disk database architectures is diminishing. Essentially, in any parallel architecture there are going to be nodes with two kinds of capabilities, namely *processing* and *I/O*. Nodes that have both capabilities would have separate processors for each. Nodes will be connected by means of a high bandwidth and low latency network, whose specific topology will be largely irrelevant. We believe that software architectures for parallel databases must keep in mind these trends in parallel hardware architecture.

PADMA has been an ongoing project for the last three years. We provide a brief overview of the project here. A detailed description can be found in [24]. Figure 2 shows the architecture of the PADMA parallel database

[1] Experience with I/O intensive applications has shown that CPU-controlled I/O is not a good idea, and a DMA or I/O processor is certainly needed [3]. Trends in architecture are similar, i.e. MPPs have dedicated I/O processors, while nodes in NOWs with I/O capability have DMAs. Coupled with the fact that NOWs will have high speed networks, accessing a remote node's disk is going to be comparable to that of accessing a disk connected to an I/O processor in a MPP.

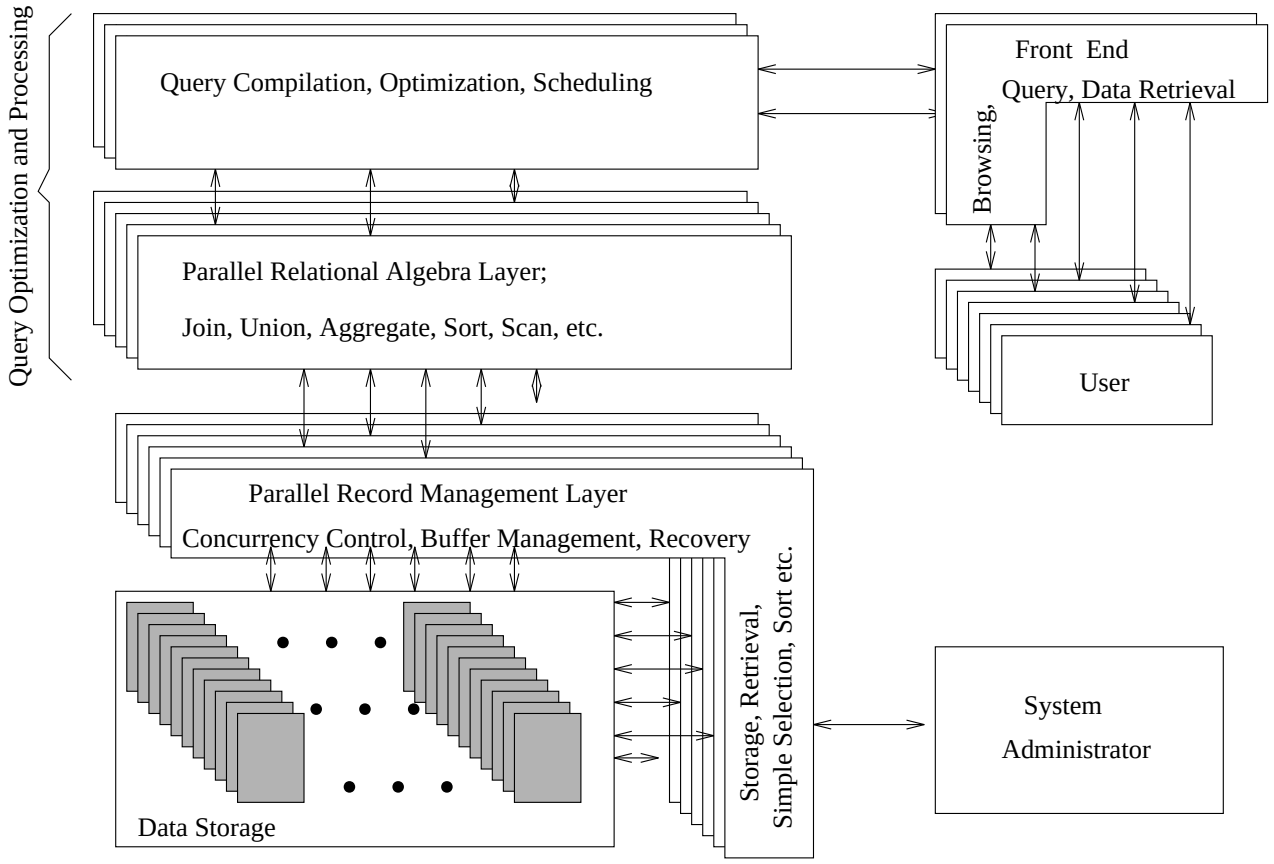


Figure 2: Parallel Database Architecture

manager. The architecture is designed around the following hypotheses:

- A geometric model, where points represent tuples, subspaces (boxes) represent queries, and geometric intersection algorithms handle query processing, is a useful way to visualize relational database querying.
- Given that data movement, between disk and memory, as well as between memories, is the main bottleneck, effective data declustering is the key to performance.
- Making the various layers of the database manager, e.g. record management, query processing and optimization, etc., understand the declustering below can have significant payoffs for performance. Thus, the DBMS layers must become *declustering-aware*.
- Multi-threading of various DBMS functions is important for performance. Additionally, the mapping of various threads to processors must be done in a *declustering-aware* manner to take advantage of the *affinity* certain processors may have for certain computation (e.g. due to data availability).

One way of viewing the results obtained in the PADMA project is as an ongoing experiment in testing the hypotheses listed above. As of this reporting the experiment is not complete. One or more of the above hypotheses have been tested to varying degrees. Others are part of our ongoing and future investigations.

The results [24] so far include (i) development of declustering techniques [14, 12] and their performance evaluation [11], (ii) declustering-aware query processing algorithms [18], (iii) parallel database loading algorithms [17], and (iv) parallel query optimization [23, 13]. We are currently building a main-memory prototype of the PADMA system.

Future work in the PADMA project includes (i) detailed performance evaluation of various techniques developed, (ii) extension of the parallel techniques developed for points to handle intervals and regions, for temporal and spatial data, and (iii) development of example applications on top of the prototype [22].

PADMA represents the effort of various individuals over the last three years. We would like to acknowledge the contributions made by Prof. Jian-Zhong Li of Heilongjiang University, P.R.C., Dr. Doron Rotem of Lawrence Berkeley Laboratory, Sakuntala Kavuri of Intel Corporation, Gary Elsseser of the University of Minnesota, and Sujal Parikh of CDAC, India. We would also like to thank the anonymous referees for their valuable comments. This research has been supported in part by the National Science Foundation grant IRI-9110584. Technical reports related to PADMA can be obtained by anonymous ftp from *ftp.cs.umn.edu: /users/padma*.

References

[1] H. Boral and et al. Prototyping Bubba: A highly parallel database system. *IEEE Transactions on Knowledge and Data Engineering*, 2(1), March 1990.

[2] D. J. DeWitt and J. Gray. Parallel database systems: The future of high performance database systems. *Communications of the ACM*, 35(6):85–98, June 1992.

[3] D.J. Dewitt and et al. The Gamma database machine project. *IEEE Transactions on Knowledge and Data Engineering*, 2(1), March 1990.

[4] H.C. Du and J.S. Sobolewski. Disk allocation for cartesian product files on multiple disk systems. *ACM Transactions on Database Systems*, pages 82–101, March 1982.

[5] C. Faloutsos and P. Bhagwat. Declustering using fractals. *Parallel and Distributed Information Systems*, pages 18–25, January 1993.

[6] C. Faloutsos and D. Metaxas. Disk allocation methods using error correcting codes. *IEEE Transactions on Computers*, pages 907–914, August 1991.

[7] O. Frieder. Multiprocessor algorithms for relational-database operations on hypercube systems. *IEEE Computer*, November 1990.

[8] T. Fujiwara, M. Ito, T. Kasami, M. Kataoka, and J. Okui. Performance analysis of disk allocation method using error correcting code. *IEEE Transactions on Information Theory*, pages 379–384, March 1991.

[9] S. Ghanderharizadeh and D.J. DeWitt. A multiuser performance analysis of alternative declustering strategies. *Proceedings of Data Engineering Conference*, Feb 1990.

[10] L. Harada, M. Nakano, M. Kitsuregawa, and M. Takagi. Query processing method for multi-attribute clustered relations. *Proceedings of International Conference on VLDB*, pages 59–70, August 1990.

[11] B. Himatsingka and J. Srivastava. Performance evaluation of grid based multi-attribute record declustering methods. *Proceedings of 10th International Conference on Data Engineering*, Feb 1994.

[12] B. Himatsingka, J. Srivastava, J. Li, and D. Rotem. Latin hypercubes: A class of multidimensional declustering techniques. Technical Report TR 94-05, University of Minnesota, Minneapolis, Department of Computer Science, January 1994.

[13] B. Himatsingka, J. Srivastava, and Thomas M. Niccum. Tradeoffs in parallel query processing and its implications for query optimization. *Technical Report TR 94-09, University of Minnesota, Minneapolis*, January 1994.

[14] Li Jianzhong, J. Srivastava, and D. Rotem. CMD: A multidimensional declustering method for parallel database systems. *Proceedings of International Conference on VLDB*, August 1992.

[15] A. S. Abdel-Ghaffar Khaled and Amr El Abbadi. Optimal disk allocation for partial match queries. *ACM Transactions on Database Systems*, pages 132–156, March 1993.

[16] M.H. Kim and S. Pramanik. Optimal file distribution for partial match queries. *Proceedings of ACM SIGMOD*, pages 173–182, June 1988.

[17] J. Li, D. Rotem, and J. Srivastava. Algorithms for loading parallel grid files. *Proceedings of ACM SIGMOD*, May 1993.

[18] Thomas. M. Niccum, J. Srivastava and J. Li. Declustering Aware Parallel Join Algorithms. *Proceedings of the International Conference for Young Computer Scientists*, Beijing, 1993.

[19] J. Nievergelt, H. Hinterberger, and K.C. Sevcik. The grid file: an adaptable, symmetric multikey file structure. *ACM Transactions on Database Systems*, pages 38–71, March 1986.

[20] David A. Patterson. A quantitative case for networks of workstations (NOW). *Cray Distinguished Lecture Series*, April 1994.

[21] H. Pirahesh and et al. Parallelism in relational database systems: Architectural issues and design approaches. *Proceedings of 2nd International Symposium on Database in Parallel and Distributed Systems, Dublin, Ireland*, July 1990.

[22] J. Srivastava. A parallel data management system for large-scale NASA datasets. *Proceedings of the 3rd NASA GSFC Conference on Mass Storage Systems and Technologies*, pages 283–299, October 1993.

[23] J. Srivastava and G. Elsesser. Optimizing multi-join queries in parallel relational databases. *Proceedings of the 2nd International conference on Parallel and Distributed Information Systems*, pages 84–92, January 1993.

[24] J. Srivastava, T. M. Niccum and J. Srivastava. PADMA: A PArallel Database MAnager. Technical Report TR 94-47, University of Minnesota, Minneapolis, Department of Computer Science, August 1994.

[25] M. Stonebraker. The case for shared nothing. *Database Engineering*, 9(1):4–9, 1986.

[26] P. Valduriez. Parallel database systems: The case for shared something. *Proceedings of 9th International Conference on Data Engineering*, pages 460–465, 1993.

[27] Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, and Klaus Erik Schauser. Active messages: a mechanism for integrated communication and computation. Technical Report TR CSD-92-675, University of California, Berkeley, Computer Science Division, June 1992.

[28] Yvonne Zhou, Shashi Shekhar, and Mark Coyle. Disk allocation methods for parallelizing grid files. *Proceedings of 10th International Conference on Data Engineering*, Feb 1994.

Fault Tolerance Issues in Data Declustering for Parallel Database Systems

Leana Golubchik Richard R. Muntz
UCLA Computer Science Department

Abstract

Maintaining the integrity of data and its accessibility are crucial tasks in database systems. Although each component in the storage hierarchy can be fairly reliable, a large collection of such components is prone to failure; this is especially true of the secondary storage system which normally contains a large number of magnetic disks. In designing a fault tolerant secondary storage system, one should keep in mind that failures, although potentially devastating, are expected to occur fairly infrequently; hence, it is important to provide reliability techniques that do not (significantly) hinder the system's performance during normal operation. Furthermore, it is desirable to maintain a reasonable level of performance under failure as well. Since high degrees of reliability are traditionally achieved through the use of duplicate components and redundant information, it is also reasonable to use these redundancies in improving the system's performance during normal operation. In this article we concentrate on techniques for improving reliability of secondary storage systems as well as the resulting system performance during normal operation and under failure.

1 Introduction

Maintaining the integrity of data and its accessibility are crucial tasks in database systems. Consequently, the reliability requirements of a database, and especially its storage hierarchy, are very stringent. A measure of a storage system's *reliability* is the mean time till it experiences loss of data due to the failure of one or more of its components; because we are interested in a continuously operating system, we use the term "data loss" to refer to inability to access data, due to failure, whether or not it is recoverable from archival storage and/or logs. (Thus, in this article we do not discuss recovery of information through the use of full dumps, log files, etc.) A database storage hierarchy typically contains a large number of disks, which not only provide the necessary storage but also the bandwidth and/or the number of disk arms required to exhibit reasonable performance[1]. For instance, transaction processing systems (i.e., with OLTP workload) are under stringent system responsiveness requirements, e.g., 99 percent of all transactions must be completed within 1 second. Such systems are configured according to the number of I/Os/second desired, rather than the number of MBs necessary to store the data [16]. Thus, the workload of the system greatly influences its storage configuration as well as (and we shall see this later) the design of its reliability schemes.

Although a single disk can be fairly reliable, given a large number of disks, the probability that one of them fails can be quite high. For example, if the mean time to failure (MTTF) of a single disk is $200,000$ hours, then the MTTF of some disk in a 200-disk system is on the order of 1000 hours, i.e., a disk failure is expected (approximately) once every 42 days. To improve the reliability and availability of the secondary storage system,

[1] A database storage hierarchy can also include a tertiary store; however, in this article, we limit our discussion to the reliability of a two level storage hierarchy.

some form of data redundancy must be introduced. One way to introduce redundancy into the system is to use parity based schemes [31] which construct a parity block for every d data blocks; the parity block plus the d data blocks constitute a parity group. Whenever a disk fails, a data block on the failed disk can be reconstructed by reading and computing the exclusive OR of the corresponding parity and data blocks. Examples of parity based schemes include RAIDs [31], clustered RAIDs [30] and various parity striping schemes [16]. Proper reliability techniques can increase mean time to data loss (MTTDL) to millions of hours [21].

Full mirroring [2] is a special case of parity striping (with $d = 1$), where each disk is replicated on another disk; whenever a disk fails, its mirror can be used to retrieve the missing data. The disk farm is composed of a number of such pairs. Full mirroring has a higher storage overhead than other parity based schemes with $d > 1$ (e.g., RAID) because data is fully duplicated, but it can offer better performance in terms of throughput and response time [16] than the parity based schemes. For instance, in [5], the authors exploit the availability of two copies of the data to optimize seek times in a mirrored disks environment.

The amount of redundant information stored determines the storage overhead for providing reliability and the system's resiliency to disk failure. The storage overhead for parity based schemes is $\frac{1}{d+1}$ of the total storage space. In general, the more redundant information is stored, the lower is the probability that a failure results in data loss, but the higher is the cost of providing reliability. Furthermore, the placement of the redundant information on the disks [24, 25] influences the system's behavior during normal operation and under failure as well as its ability to recover quickly and return to the fully operational state. When designing a fault tolerance scheme, the following aspects of the disk subsystem must be examined: a) performance under normal operation (e.g., [7, 31]), b) mean time to data loss (or system failure) (e.g. [14]), and c) performance of the disk subsystem under failure, i.e., when one or more disks are inoperable or inaccessible (e.g., [35, 30, 17, 19, 18]). We should keep in mind that failures are expected to occur relatively infrequently, so most of the time a system is in a fully operational mode. Thus, it is important to provide reliability techniques that do not (significantly) hinder the system's performance during normal operation. Since high degrees of reliability and availability are achieved through the use of redundant information (and duplicate components), it is also reasonable to use these redundancies in improving the system's performance during normal operation, e.g., as in mirrored disk systems [5] (see Section 3 for more details).

In order to maintain a reasonable MTTDL, it is desirable to provide immediate repair of a failed disk; after the first failure has occurred, there is a vulnerability window during which a second failure causes loss of data. (We assume, as is true of all the schemes surveyed here, that for every disk failure, the additional failure of one of the surviving disks can cause data loss.) To this purpose, "hot standby" disks (or spares) are often provided, and the system is designed to automatically rebuild the contents of the failed disk on the standby disk, using the redundant information on the surviving disks [14]. The parity group size effects: a) the time required to rebuild a failed disk (and therefore the MTTDL) and, b) the workload (measured in accesses per second per disk) that can be supported during the rebuild process, and c) the system's performance under failure. The MTTDL of a RAID is easily shown to be inversely proportional to the rebuild time [6, 11]; in the RAID system described in [31], rebuilding the failed disk contents at maximum speed (the capacity of the standby disk) results in the use of the entire capacity of the surviving disks in the array. Thus rebuilding at maximum rate means that the array can perform no other work during the rebuild period. One can of course, tradeoff the rebuild rate with the rate at which the surviving disks process normal workload requests. However, this increases the time to rebuild the failed disk contents and thereby decreases the MTTDL.

Although disk failures are infrequent, a single disk unavailability is still a relatively common occurrence as compared to data loss (or system failure). Therefore the performance of the system under failure and especially during the repair period (when the data on a failed disk is being rebuilt) is of concern. The RAID organization achieves a low cost in redundant storage overhead, as compared to mirrored systems, but at the price of degraded performance under failure [2]. In the worst case (a workload of all reads and no writes) this can double the access

[2] Note that, RAID systems also pay a performance penalty during normal operation; this is due to having to write a parity block on every write operation.

rate to the surviving disks and thus in effect, cut the capacity of the array in half. Consider for example a shared-nothing [34] database machine architecture, where each node contains one or more disk arrays. The impact of a failure on the total system performance is dependent on the characteristics of the system workload; it is most severe in the case of a "decision support" environment in which complex queries are common and the database tables have been partitioned among the disks on all or many nodes, for the purpose of increasing I/O bandwidth. Such complex operations are typically limited by any imbalance[3] in the system, which can be caused either by a skew in the workload [23] or by a disk array with a diminished capacity, due to a failure [20]. For example, in a one hundred disk system, a single failed disk represents a loss of only 1% of the raw I/O capacity of the system. However, if the effect of the failure is a reduction in the capacity of the array (to which it belongs) by say 25%, then this failure can cause a significant imbalance in the system, and the impact on aggregate system performance can be considerable.

In this article, we discuss techniques for providing a high degree of reliability and availability in a database system; these techniques can be divided into two basic categories, which are as follows: 1) full replication, which includes schemes such as shadow disks, interleaved declustering, and chained declustering, and 2) parity based redundancy, which includes schemes such as RAID (Redundant Arrays of Inexpensive Disks), clustered RAID, and parity striped disk arrays. We also discuss the tradeoffs, associated with each of these techniques, with respect to the following metrics: a) storage overhead (due to redundancy), b) mean time to data loss (MTTDL), c) performance during normal operation, and d) performance under failure.

The remainder of the article is organized as follows. Section 2 points out the differences between physical and logical replication. Section 3 discusses full replication schemes, and the advantages and disadvantages associated with those, both in the context of reliability and performance. Section 4 presents a similar discussion, but in the context of parity based schemes. Finally, Section 5 presents our concluding remarks.

2 Physical vs. Logical Redundancy

In general, data redundancy can be implemented on different levels within a database system. In particular, we distinguish between (1) physical redundancy and (2) logical redundancy. In what follows, we discuss the differences between physical and logical redundancy in the context of full replication schemes; however, similar comments apply to parity based schemes, such as RAID systems.

With physical level replication the contents of one area of a disk are mirrored on an area of another disk (in the classical mirrored disk system, one entire disk is mirrored by another entire disk). The I/O controller generally handles the replication and higher levels of software, such as the query optimizer, are not concerned[4], i.e., higher levels of software just see a collection of reliable disks with some changes in performance characteristics. With logical fragmentation as in the Teradata [3] and Gamma [12] shared nothing database machines, relations are fragmented and relation fragments are stored on independent nodes of the system. Replication is visible to the query processing software and is managed by the database system software. For instance, since read requests can be serviced using either copy of the data, replication can be used for load balancing purposes (we elaborate on this further in Section 3). For reads, load balancing decisions can be made by the query processing software, i.e., at the logical level[5], as in [20], or they can be deferred until the time of the actual I/O operation, i.e., performed by the disk controller at the physical level, as in [15].

Note that, the dynamic scheduling studies that are discussed in this article, specifically in the context of chained declustering (see Section 3.3), can be applied to both physical and logical replication methods. There

[3] Such queries would typically be performed in a "fork-join" manner (on a shared-nothing machine), where the performance is limited by the "slowest" node participating in the computation.

[4] Similarly, in RAID systems, higher levels of software just see a disk, i.e., a (logical) reliable disk with some changes in performance.

[5] Note that, one can have logical level replication and *not do* dynamic load balancing, i.e., just use the replication for reliability and (static) redistribution of load after failure.

are however significant problems associated with dynamic data sharing across multiple nodes of a system, e.g., concurrency control, and efficient use of buffer space [38, 37]. We do not address these problems here due to lack of space. With respect to logical replication one can view such studies as an investigation of the *potential* benefits of dynamic load balancing, particularly with respect to robustness to workload imbalance and disk failure. Determining whether these benefits compensate for the overhead and complexity of logical level dynamic scheduling is beyond the scope of this article. In the remainder of this article we will concentrate mainly on physical replication, with the exception of interleaved and chained declustering schemes[6] discussed in Sections 3.2 and 3.3, respectively.

3 Full Replication

We first concentrate on systems that use full replication as a form of redundancy, and present three variations on this idea: 1) mirroring or disk shadowing, 2) interleaved declustering, and 3) chained declustering. Since all three schemes fully replicate the data, they differ only in the way the replicas are placed on the disks. This placement affects both reliability and performance.

3.1 Mirroring/Shadowing

Disk shadowing [5, 2] refers to maintaining two (*mirrored disk*) or more (*shadow set*) identical disk images on different disks, mainly for the purpose of providing a highly reliable disk subsystem. A read request to the shadow set can be satisfied by any disk in the set; a write request must be executed on each of the disks in the shadow set. When a disk fails, the data is still available on the other disks in the shadow set. To replace the failed disk, the data must be copied from one of the disks in a shadow set to a replacement disk. This can be done either offline or online. Offline copying is fast, but requires losing availability of data during the copying process (this can be on the order of minutes/GB). Online copying has the advantage of availability of data but can be much slower than offline copying (on the order of several hours). During the copying process the disk subsystem is vulnerable to a second failure; with only two disks in a shadow set, a second failure results in data loss. Furthermore, the system operates at a degraded level of performance. This degradation in performance is due not only to the failure of a disk, but also to the copying process, which results in an additional workload on the shadow set. The more "aggressive" is the copying process, the more it interferes with the normal workload. However, the faster a failed disk is replaced, the less likely we are to lose data and the shorter is this degraded mode of operation. Hence, it is desirable to balance the speed of the copying process, with degradation of performance experienced by the normal workload due to the copying.

There are several disadvantages to disk shadowing. Firstly, there is the cost. Mirroring has a 100% storage overhead. This is not a severe problem if the expected workload is of the OLTP type. According to [16], OLTP systems have stringent responsiveness requirements; in order to avoid long queues of requests for the data, the disks in such systems are usually purchased for their arms and not for their capacity. Secondly, there is the "write" overhead. Since a write request must be serviced by every disk in a shadow set, it is not complete until the last disk has finished writing. Even if all the disks in a shadow set can start working on the request simultaneously, the write request will still experience the largest value of seek-plus-latency of all the disks in the shadow set.

There are advantages to disk shadowing, besides high reliability, which should be considered when comparing its cost to the cost of parity based schemes. One such advantage, perhaps not an obvious one, is performance. With multiple data paths, a shadow set can service several read requests in parallel, thus improving the throughput of the disk subsystem. Furthermore, expected seek times for read requests can be improved by chosing the disk in the shadow set with the minimum seek distance [5, 4]. This leads to a need for disk scheduling policies to exploit these possibilities. Such policies for mirrored disk subsystems are studied in [36]; disk scheduling policies for

[6] These schemes were originally suggested as logical level schemes; thus we discuss them in that context.

real-time applications using mirrored disks are studied in [9]. One interesting question that is addressed in [5] is whether it makes sense to have more than 2 disks in a shadow set. The authors argue that two copies are sufficient to provide a high degree of reliability, but that more than two copies can result in significant performance improvements.

3.2 Interleaved Declustering

In [3, 11] interleaved declustering is considered as a replication scheme at the logical level (see Section 2). It can also provide an alternative to the mirroring[7] scheme, if applied at the physical level. We briefly describe this scheme, which is illustrated in Figure 1, applied to physical level replication. The secondary storage subsystem

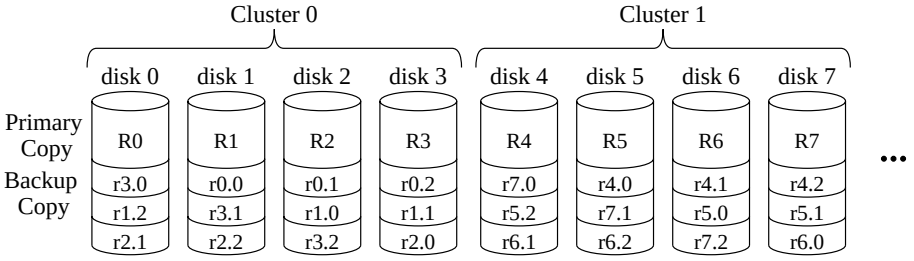


Figure 1: Interleaved Declustering

is divided into disk clusters, each of size N, e.g., in Figure 1, $N = 4$. Each file or table, R, is allocated equally to each cluster; then each part assigned to a cluster is divided into N fragments. At all times two copies of this file or table exist, termed primary copy and backup copy; both copies reside on the same cluster. The primary copy of each fragment resides on one of the disks in a cluster, and the backup copy of the same fragment is divided equally among the remaining $N - 1$ disks of the cluster. During normal operation, read requests are directed to the primary copy[8] and write requests are directed to both copies (as in the mirrored disks case). When a failure occurs, for instance of disk 1 in Figure 1, the read workload that was destined for disk 1 can be distributed among the surviving $N - 1$ disks of the cluster in which the failure occurred. This is an improvement over the mirrored disks scheme where the additional workload, that was destined for the failed disk, ends up on a single surviving disks (i.e., mirroring is a special case of interleaved declustering with $N = 2$).

Thus, interleaved declustering has the same storage overhead as mirroring, but it offers better performance degradation properties, when a single disk failure occurs. The larger the cluster size, the smaller is the imbalance in the workload (in the event of failure) between the fully operational clusters and the cluster with a failure[9]. However, as the cluster size increases, so does the probability of two failures in the same cluster. Two failures in any one cluster render data unavailable. Hence, the use of mirrored disks offers a higher level of reliability than interleaved declustering (i.e., schemes with $N > 2$)[10].

[7] In the remainder of Section 3 we make comparisons to the mirrored disk scheme (i.e., shadow sets with 2 disks only), since it incurs the same storage overhead as interleaved declustering and chained declustering (discussed in Section 3.3).

[8] Note that, it is possible to use both copies of the data to service read requests; however, with logical level replication, concurrency control and buffer management issues must be considered (see Section 2).

[9] This could be a significant problem, for instance, in a shared-nothing database machine (see Section 1), such as the DBC/1012 [3], where the performance of the "slowest" node limits the performance of the entire system.

[10] Note that, this argument is an approximation, i.e., it only takes into consideration combinations of 2 failures. To make precise calculations, we must take into consideration combinations of 3 or more failures; however, these are much less probable than combinations of 2 failures.

3.3 Chained Declustering

In [20], chained declustering is considered as a replication scheme at the logical level of a shared nothing database machine. This scheme can also provide an alternative to the classical mirroring scheme when applied to physical level replication, as well as to the interleaved declustering scheme described in [3, 11]. We briefly describe the concept of chained declustering from [20].

Chained declustering has the same storage overhead as compared to the classic mirroring scheme and interleaved declustering, but, like interleaved declustering, it offers better performance degradation properties when a single disk failure occurs. Figure 2 illustrates the chained declustering concept. Assume a file R is declustered

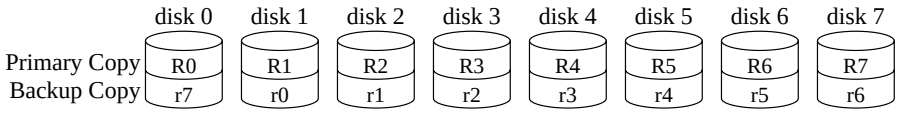


Figure 2: Chained Declustering

into M fragments, where M is the size of a disk cluster (e.g., in Figure 2, $M = 8$). At any point in time, two physical copies of this file, termed the primary copy and the backup copy, are maintained. If the primary copy of a fragment resides on disk i, then the backup copy of that fragment resides on disk $i + 1 \pmod M$. During the normal mode of operation, read requests are directed to the primary copy[11] and write operations update both copies. When a disk failure occurs (e.g. disk 1 in Figure 2), the chained declustering scheme is able to adjust the additional read workload to both copies of the data in such a way as to balance it evenly among the surviving disks; this results in a less degraded performance (see [20] for more details).

There are several ways to perform the load adjustment depending on table declustering methods, storage organization, and access plans. Since data is logically replicated, the query scheduler chooses an access plan in order to balance the load. This form of load balancing has several limitations: (1) the load is only approximately divided among the nodes; the assumption that a uniform division of the data corresponds to a uniform division of the load can be incorrect with skewed reference patterns and (2) both short term and long term reference patterns change with time and a static balancing scheme can not adjust to variations in load. Another way to balance the load of the system is to apply some *dynamic* load balancing scheme, since it can adjust the load on each node in real time to respond to statistical variations[12]. As already mentioned, several dynamic balancing schemes are discussed in [36], in the context of mirrored disks systems. In [15], authors investigate the degree to which a dynamic load balancing disk scheduling algorithm in conjunction with chained declustering can respond robustly to variations in workload and disk failures (which destroy the symmetry of the system and introduce skewed load); they demonstrate that simple dynamic scheduling algorithms can greatly improve the average response time as compared to static load balancing.

Chained declustering has the same storage overhead as mirroring and interleaved declustering. But, it has a higher reliability than interleaved declustering (but not as high as mirroring) [20]. In order to lose data in the chained declustering scheme (refer to Figure 2), two *consecutive* disks in the same cluster must fail. Note that the probability of two consecutive disks failing in the same cluster, for $M > 2$, is independent of the size of the cluster. Hence, in the case of chained declustering, constructing a single cluster out of all the disks in the system does not hinder the system's reliability, but it can offer better load balancing in the event of failure. Since there is no reliability penalty for using large clusters, the increase in load, due to a failure, can be made as small as desired by increasing the cluster size. This is not the case for interleaved declustering (as already mentioned in Section

[11] As with interleaved declustering, it is possible to use both copies of the data to service read requests; however, with logical level replication, concurrency control and buffer management issues must be considered (see Section 2).

[12] Dynamic load balancing would result in additional complexity in query processing software (see Section 2), e.g., in terms of concurrency control; such complexity can be expensive, and consequently, dynamic load balancing schemes might be more suitable for large queries, such as found in decision support type workloads, rather than OLTP type workloads.

3.2). Thus, chained declustering (for $M > 2$) offers better load balancing than either mirroring or interleaved declustering, since it is able to distributed the additional load (due to failure) among *all* the disks in the storage subsystem as opposed to a single disk (as in the case of mirroring) or the disks in a single cluster (as in the case of interleaved declustering)[13].

4 Parity Based Schemes

As already mentioned in Section 3, full replication schemes have the disadvantage of a 100% storage overhead. To remedy this problem, we can use a parity based scheme. In this section we discuss three variations on such schemes: 1) redundant array of inexpensive (or independent) disks (RAID) [31], 2) clustered RAID [30], and 3) parity striping [16]. We should note that there exists another variation on the RAID idea, termed RADD (Redundant Array of Distributed Disks), which is a distributed version of a RAID5 system (refer to Section 4.2 for a discussion on RAID5); we do not discuss it here due to lack of space but refer the interested reader to [35].

4.1 Disk Array Basics

The basic organization of an $N+1$ disk array is illustrated in Figure 3, where there is a cluster of $N+1$ devices with N data devices and one parity device ($N = 3$). A file R is fragmented into blocks of size s, termed the

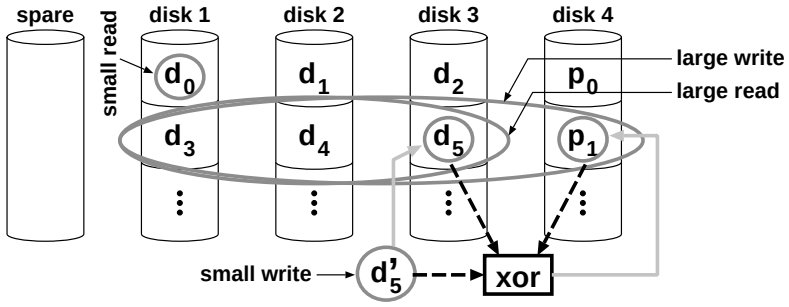


Figure 3: Basic RAID Organization

interleave unit size or the *stripe unit*, which is the amount of logically contiguous data that is placed on a single device, e.g., d_0 in Figure 3. The file is then interleaved among the N data devices, where N is the *stripe width*. Each set of N data blocks is protected by one parity block; for instance in Figure 3, $p_0 = d_0 \oplus d_1 \oplus d_2$.

In general, there are three modes of operation for a disk array [30]: 1) *normal mode*, where all disks are operational, 2) *degraded mode*, where one (or more) disks have failed, and 3) *rebuild mode*, where the disks are still down, but the process of reconstructing the missing information on spare disks is in progress. Under normal operation read requests are directed to the disks holding the appropriate data[14]. A "small" read operation would result in a single disk access, and a "large" read operation would result in a full stripe access (i.e., involving all the disks in the cluster, except the parity disk). Every write request involves an access to at least 2 disks, due to the necessary parity update. For instance, to replace d_5 by $d_5^{'}$ in Figure 3, we must read d_5 and p_1, from disks 3 and 4, respectively, then compute the new parity, $p_1^{'} = d_5 \oplus p_1 \oplus d_5^{'}$, and then write out $d_5^{'}$ and $p_1^{'}$, to disks 3 and 4, respectively. Hence, a "small" write operation, involving data on a single disk, results in 4 I/O accesses, two

[13] We should note, that although there is no reliability penalty associated with using large clusters in a chained declustering scheme, there are potential performance penalties. For instance, if the size of the fragments becomes very small (which would happen if a relatively small file was distributed over many disks), then some types of queries would have to be serviced by accessing multiple disks, and this can result in increased overhead [13].

[14] This could involve one or more disks of the array, depending the granularity of the stripe unit and the size of the request .

reads and two writes. A "large" write operation would be a full stripe access and result in a write on every disk in the cluster, i.e., there is no need to read the old parity or the old data (e.g., $p_0^{'} = d_0^{'} \oplus d_1^{'} \oplus d_2^{'}$).

After a failure occurs the system continues to operate but in a degraded mode. For instance, suppose disk 3 fails in the system of Figure 3; then, to service a read request destined for the failed disk (e.g., d_2), we must read a full stripe in order to reconstruct the missing data, (e.g., $d_2 = d_0 \oplus d_1 \oplus p_0$). To service a write request destined for the failed disk, we must do one of the following things. If the write request is for a *data* block, then we must read the full stripe, to reconstruct the missing block, compute the new parity, and write the new parity. If the write request is for a *parity* block, then it can be ignored. These additional full stripe reads and writes that are necessary to reconstruct the missing data, result in a degraded performance of the disk subsystem. Note, that the above description of servicing reads and writes destined for the failed disk is relevant to "small" reads and writes only. "Large" read and write requests are full[15] stripe operations regardless of whether there is a failure or not. (This ignores edge effects of "large" operations, i.e., $1\frac{1}{2}$ stripes, for example.)

To reconstruct the missing data, i.e., enter the rebuild mode, we need a spare disk. Having a *hot* spare, i.e., a spare disk that is online and ready for reconstruction as soon as a failure occurs, would significantly decrease the vulnerability period, i.e., the period in which another failure would result in loss of data; decreasing this period is also desirable because of the degraded system performance under failure. The basic reconstruction procedure works as follows. A full stripe is read from all the surviving disks in the cluster, including the parity block. Then the missing data block (from that stripe) is computed and written out to the spare disk. For instance, to reconstruct d_2 in Figure 3, we read d_0, d_1, and p_0, and then compute the missing data, $d_2 = d_0 \oplus d_1 \oplus p_0$. Finally, d_2 must be written out to the spare disk.

Before discussing disk arrays in more detail, we present a list of design issues which should be considered when constructing parity devices (in the following sections we address some of these issues in more detail): a)*redundancy support for hardware* in addition to redundant information, e.g., multiple controllers, b) *independence of device failure* is important since I/O subsystems require support hardware that is shared among multiple disks (see Section 4.6), c) *array size (or cluster size)* affects the reliability of the system as well as its performance in the normal and degraded modes of operation (see Sections 3 and 4.4), d) *stripe width (or parity group size)* in the traditional RAID organization (see Section 4.2) is equal to the cluster size, whereas in Section 4.4 we show how the system's performance under failure can be improved by relaxing this condition, e) *interleave unit size (stripe granularity)* determines the number of devices that are involved in an access, and and hence it affects the system's performance during normal operation (we do not discuss this any further due to lack of space but refer the interested reader to [14] for a performance comparison between byte interleaved, block interleaved, and mirrored systems under normal operation), f) *number of spares* affects the reliability of the I/O subsystem (see Section 4.3), and g) *reconstruction time (or vulnerability window)* is of crucial importance, because a system operating under failure is not only vulnerable to a second failure (which results in a system failure, i.e., loss of data) but it also exhibits degradation in performance; to reduce the MTTF of the whole system, it is necessary to rebuild the failed disk as soon as possible but without significantly slowing down the normal workload; in other words, the availability of data after a failure would not mean much if this data can not be accessed in a "reasonable" amount of time (see Section 4.5 for a discussion of several reconstruction schemes).

4.2 RAID Organizations

In this section we describe the different RAID organizations, as they are presented in [31]. Firstly, we present the terminology[16]: 1) **RAID1**: is a data mirroring scheme, i.e., it uses full replication (see Section 3), 2) **RAID2**

[15] A large read doesn't involve an access of the parity disk, under normal operation. The failure's affect on system's performance depends on the RAID organization used; e.g., there would be no impact on the performance of a RAID3, because it uses the rotationally synchronized byte interleaved organization which does not allow multiple parallel accesses anyway (see Section 4.2).

[16] We do not describe the RAID1 scheme in more detail, since it is very similar to the full redundancy schemes discussed in Section 3. The RAID2 organization uses Hamming code as its ECC, where some fraction of the redundant information is used to detect which

& RAID3: are parity based, parallel access schemes, where all the disks in a cluster are rotationally synchronized, and 3) **RAID4 & RAID5**: are parity based, independent access schemes, where all the disks in a cluster can simultaneously perform independent accesses. The synchronized RAID3 organization is traditionally byte interleaved, as in [22, 31]. This is due to the common assumption that rotationally synchronized disks do not perform independent accesses; hence, they are viewed as a single unit, with $N *$ rate of a single disk, and which can satisfy one request at a time. (An exception to this view is the work presented in [8], where the authors describe workloads under which it would be beneficial to use larger striping units in synchronized, i.e., RAID3, disk array organizations.) The advantages of a traditional byte interleaved RAID3 are: 1) high bandwidth, 2) high reliability, and 3) its performance in degraded modes (since every request results in a full stripe access, its performance in degraded mode is equivalent to its performance in normal mode). A disadvantage of RAID3 is that it has low throughput on small requests, since every request involves all the disks in cluster, no matter how large or small.

To remedy the problem of low throughput on small accesses, we can use the RAID4 and RAID5 schemes, which both use block[17] interleaving and can independently service multiple requests in parallel. The difference between the two schemes is in the parity placement[18]. In the RAID4 scheme there is a dedicated parity disk, as in the example of Figure 3. The problem with this arrangement is that the parity disk can become a bottleneck, since every small write operation requires the reading and writing of parity. To remedy this problem, the RAID5 scheme rotates the parity among all the disks in a cluster; this is illustrated in Figure 4. The basic idea

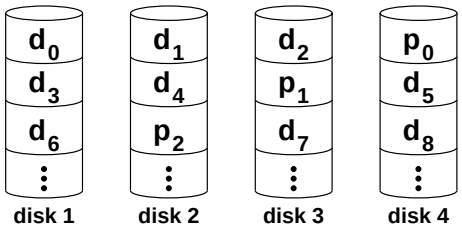


Figure 4: RAID5 Organization

is that RAID4 and RAID5 should still provide the high access rate of RAID3 on large requests but are also able to provide high throughput on small requests. However, we should note that RAID4 and RAID5 suffer from performance degradation on "small" write requests, since each non-full stripe write request results in four I/O operations; due to lack of space, we do not discuss this problem here but refer the interested reader to [27, 26, 33].

4.3 Spares

As mentioned earlier, reconstructing a failed disk as soon as possible contributes significantly to improving the MTTDL. Of course, to reconstruct a disk, we need a spare one. If the spare disk is offline, i.e., requires human intervention, then the time to order it, install it, etc. will likely dominate the actual reconstruction process. However, if the spare disk is online (i.e., a *hot* spare), then the vulnerability period[19] of the system is determined by the efficiency of the reconstruction process. The various approaches to improving the reconstruction process are discussed in Section 4.5. In this section, we first address the question of "how many spares do we need?". In [14], the authors address this issue by simulating a disk array with 7 parity groups (or strings as they are called in [14]) and varying the size of the spare disk pool. The basic result is that (with or without hot spares) there is

disk has failed (only one parity disk per cluster is necessary to correct the failure). Since most disk controllers can detect which disk has failed, this is not necessary. Thus, we do not discuss the RAID2 organization any further.

[17] What is the desirable block size depends on the system's expected workload (e.g., see [8]).

[18] Performance consequences of several parity placement schemes for RAID systems are investigated in [24, 25], where the authors show that, for certain types of workloads, a "proper" choice of parity placement can result in a significant performance improvement.

[19] The period during which another failure results in data loss.

essentially no difference (with respect to MTTDL) between a spare pool of 7 disks and an "infinitely" large spare pool, i.e., it is sufficient to provide one spare disk per parity group.

Another way to use spare disks to improve system performance, both during normal operation and under failure, is to use a distributed spare [28] (instead of a dedicated spare). The basic idea is to use the spare disk under normal operation to construct an $N+2$ (instead of an $N+1$ array) with spare blocks on all $N+2$ disks; an example of a system using a distributed spare is illustrated in Figure 5. Advantages of an array with a distributed

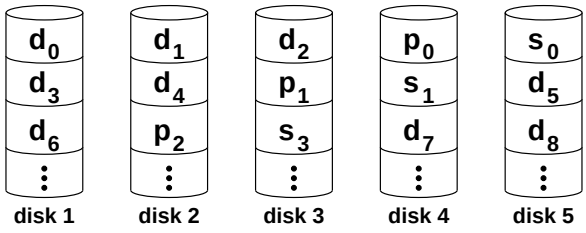


Figure 5: Distributed Spare (N+2 Array)

spare are as follows: a) better performance under normal operation, since we are able to use $N+2$ instead of $N+1$ disks, b) better degraded mode performance, since we are able to use $N+1$ instead of N disks plus less data is lost due to failure (since the spare blocks had no data), c) shorter reconstruction process (since less data is lost due to failure), and d) higher probability that the spare is operational when it is needed, since it is being used during normal operation (see Section 4.6 for discussion on infant mortality of disks). The disadvantages are: a) when a new disk becomes available (to replace the failed one), there is a need for a "copy back" process, i.e., copying of data to the new disk in order to create a distributed spare again, which could be done when the system is idle, and b) with $N+2$ disks in an array, there is a greater probability of a single disk failure, and hence, distributed spare systems tend to spend more time in degraded performance modes.

4.4 Clustered RAID

It is desirable for the system to spend as little time as possible in the degraded mode of operation, because during that period: a) the system is vulnerable to a second failure, which can result in data loss and b) the system performance is degraded due to the failure. One way to improve the system's performance under failure and at the same time speed up the reconstruction process is to use the *clustered* array organization, proposed in [30]. The basic idea behind clustered disk arrays is to relax the assumption that the group size, G, should be equal to the cluster size, C, where "group" refers to the parity groups size, i.e., the number of data blocks plus the parity block, and the "cluster" size refers to the number of disks over which the parity group blocks are distributed. In a traditional RAID architecture, as in [31], it is assumed that the group size is always equal to the cluster size. An example of a system where the group size ($G=4$) is less[20] than the cluster size ($C=5$) is illustrated in Figure 6. To place each parity group (i.e., three data blocks plus one parity block) on the disks, we must select 4 out of 5 disks in the system. Since there are $\binom{5}{4}$ ways to make such a selection in Figure 6, there are five possible types of parity groups[21].

The clustered organization does not require additional disks, since the overhead for storing redundant information is determined by the group size, G; furthermore, G determines the number of reads that must be performed to reconstruct a data block from a failed disk. On the other hand, the MTTDL is determined by the cluster size, C, since any two failures in one cluster result in data loss. Note that, there are benefits in choosing a cluster size

[20] Of course, the group size has to be at most as large as the cluster size, otherwise, the array would not able to recover even from a single failure.

[21] Note that in Figure 6, each group type appears to have a column of empty blocks; this is done for ease of illustration, i.e., the figure illustrates the *logical* organization of the data on the disks rather than the *physical* one.

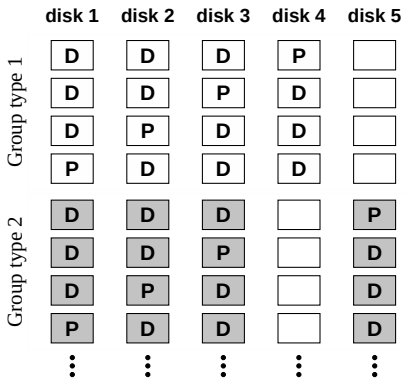


Figure 6: Clustred RAID Organization

that is greater than the corresponding group size, and they are as follows. When a disk fails, $G-1$ blocks must be read, from $C-1$ surviving disks in order to reconstruct each block of the missing data. By properly distributing the groups among all the disks in the cluster, the additional load, due to failure, can be distributed evenly over all $C-1$ surviving disks. If r is the fraction of accesses in the normal workload that are reads, then the increase in the workload due to one failed disk is determined by $r\frac{G-1}{C-1}$. Hence, an array with $G < C$ would perform better under failure and would have a shorter reconstruction process. An analysis of clustered array's performance under failure, using three different reconstruction schemes (see Section 4.5) can be found in [30]; this analysis indicates that there are significant advantages to using the clustered disk array scheme. There remains one problem with respect to implementing the clustered array architecture, which is left open in [30]. This is the problem of computing, for a given data block, the location of its "buddy" data blocks and parity block (i.e., the rest of the blocks in the parity group), which is addressed in [17, 29].

4.5 Recovery Procedures

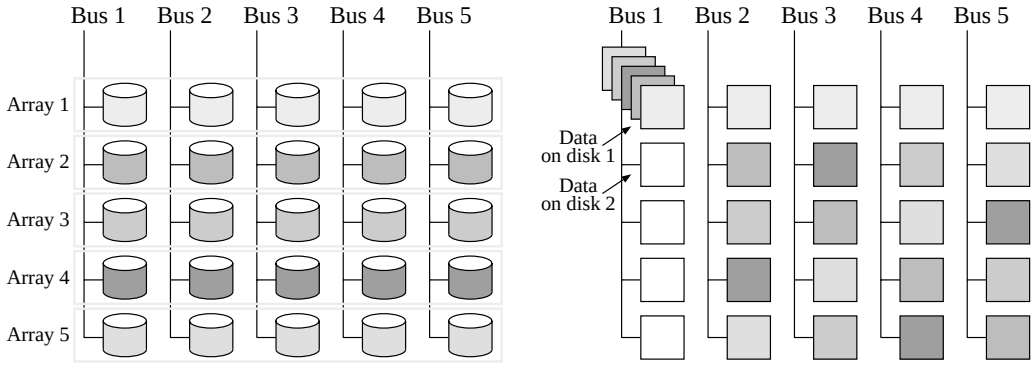
Several reconstruction schemes are suggested in [30]; these include: a) *basic* rebuild, where the data is read from the surviving disks, reconstructed through a parity computation, and then written to the spare disk, b) rebuild with *read-redirect*, where, in addition, read requests, for the portion of the data on the missing disk that has already been reconstructed on a spare, are redirected to the spare disk, and c) *piggy-backing* rebuild, which takes advantage of read requests for data on surviving disks and uses the retrieved information to reconstruct some portion of the failed disk. In all three schemes, the authors [30] suggest that the write requests to the failed disk should *always* be redirected to the standby disk. In [17] the authors question this decision and investigate another recovery algorithm, in addition to the three proposed in [30], which they refer to as the *minimal-update* algorithm; in this scheme, updates to the failed disk are ignored, whenever possible. A simulation of all four reconstruction algorithms reveals that the two more complex schemes, i.e., read-redirect and piggy-backing, do not consistently reduce the length of the reconstruction period. In particular, in light to moderate loads with $\frac{G-1}{C-1} < 0.5$, the schemes with no redirection result in a shorter reconstruction period. The reason [17] is that the benefits of offloading the surviving disks do not outweigh the penalty of loading the replacement disk with *random* workload, unless the surviving disks are highly utilized.

Several other issues should be considered when designing a reconstruction process, for instance, the size of the reconstruction unit, which can be a track, a sector, a cylinder, etc. The tradeoffs are as follows. A larger reconstruction unit should speed up the reconstruction process, however, it should also result in greater degradation of performance, as experienced by the normal workload, i.e., the longer it takes to read a reconstruction unit, the (possibly) greater is the queueing delay experienced by the normal workload. Another way to reduce the reconstruction period is to start multiple (independent) reconstruction processes in parallel. In [17], the authors note

that a single reconstruction process (or in lock step reconstruction)[22] is not always able to highly utilize a disk array, especially when $\frac{G-1}{C-1}$ is relative small; in that paper, the authors investigate the benefits of using an 8-way parallel reconstruction process[23].

4.6 Independence of Disk Failures

Until now, we have primarily considered the failure of disks. However, there are other components in the I/O subsystem that deserve attention, such as controllers, power supplies, cabling, cooling systems, etc. In [14], the authors point out that such support hardware is normally shared by a *disk string* (all the disks on one bus), as illustrated in Figure 7(a). A failure of one such shared hardware component, e.g., a power supply, would result



(a) Orthogonal Parity Group Organization (b) Disk Matrix

Figure 7: Independence of Disk Failure

in the inaccessibility of an entire string of disks. Thus, disks sharing the same support hardware should not belong to the same disk array. In fact, the disk arrays should be constructed orthogonally to the support hardware groups [32]. In [14], the authors compare the MTTDL of an array with an non-orthogonal organization to that of an array with an orthogonal organization and show a significant improvement in reliability.

In addition to guarding against multiple failures due to a single support hardware failure, we would also like to have an even load distribution, over all the disks in the system, when a failure does occur. However, the orthogonal organization described above does not exhibit this property. Note that in that organization, a disk failure creates an addition load *only* on the disks belonging to the same disk array as the failed disk. In [1], the authors propose another approach, termed a *disk matrix*[24], which also guards against single points of failure but with an additional benefit of evenly distributing the additional load due to a failure over all the disks in the system. In general, all blocks belonging to the same parity group (i.e., data blocks plus a parity block) are distributed among the disks in the disk matrix according to the following rules: 1) no two blocks from the same parity group end up on the same disk string and 2) the increase in the load due to a disk failure is evenly distributed among all the disks in the matrix [1]; this is illustrated in Figure 7(b). Due to a lack of space we do not describe this scheme any further but refer the interested reader to [1]. We do note, however, that one disadvantage of this scheme, as compared to the orthogonal organization scheme, is that it has a lower reliability, since essentially, it uses larger clusters.

[22] By a single reconstruction process (or in lock step reconstruction) we mean a recovery procedure where the reconstruction of one data block must be completed before the reconstruction of another data block can begin.

[23] The parallel reconstruction process requires additional buffer space to hold the data blocks that have been read from the surviving disk, but have not (yet) been used to reconstruct the missing data.

[24] The disk matrix is a generalization of the clustered disk array idea.

4.7 Parity Striping

In [16], the authors point out why traditional RAID5 organization [31] might not be the best solution for all types of workloads, and more specifically for OLTP workloads (i.e., workloads with relatively small accesses). The reason is that OLTP systems can not afford to use several disk arms on a single transfer, because the reduction in (an already fairly short) transfer time can not offset the overhead associated with parallel transfer, such as an increase in seek plus latency time (due to using multiple arms). Therefore, the authors propose another striping scheme, termed *parity striping*, which can provide cheap reliable storage and high throughput. The basic idea behind parity striping is to make a $N + 2$ disk array look like $N + 1$ logical disks plus a spare disk, rather than as one logical disk (as in a RAID architecture). To this end, only parity blocks (rather than files) are striped across all the disks in the system. Such a system is illustrated in Figure 8, where, for instance, blocks p_{20} and p_{21} represent one contiguous parity segment which holds parity information for data blocks d_{00}, d_{01}, d_{10}, and d_{11} (where blocks d_{0i} belong to file 0 and blocks d_{1i} belong to file 1). Thus, a parity striping architecture allows

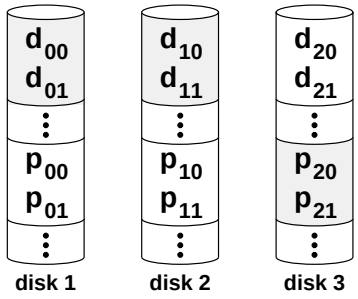


Figure 8: Parity Striping

each small (relative to the size of the parity segment) access to be satisfied by a single disk, but it still provides the reliability of a RAID5 system. In [16], a system using parity striping is analyzed and its performance is compared to that of a system using mirrored disks and a system using RAID5. Another comparison of RAID5 and parity striping performance (under normal operation) can be found in [10]. Due to a lack of space, we do not discuss these works here.

5 Summary

In summary, we have discussed two basic categories of schemes that store redundant information for the purpose of reliability; these are: 1) full replication schemes and 2) schemes using parity information. In general full replication schemes exhibit higher reliability and better throughput under normal operation (if both copies of the data are used to service read requests). On other other hand, schemes using parity information have a much lower storage overhead. The reliability characteristics of (most) schemes presented in this article can be summarized briefly, as follows. To lose data in a system with D disks, the following must happen: 1) with **mirroring** two disks must fail in the same mirrored pair, and there are $\frac{D}{2}$ such combinations, 2) with **interleaved declustering** two disks must fail in the same cluster, and there are $\binom{C}{2}\frac{D}{C}$ such combinations, where C is the size of each cluster, 3) with **chained declustering** two *consecutive* disks in the same cluster must fail, and there are $C = D$ such combinations, where C is the size of the cluster (recall, that in chained declustering there is no reliability penalty due to larger clusters, and in addition, there is a benefit to having larger clusters, namely the reduction in additional load due to failure. Thus, it is (usually) desirable to have all the disks belong to the same cluster; hence, $C = D$ above), 4) with **traditional RAID** two disks must fail in the same cluster of size $C = G$ (where G is the parity group size), and there are $\binom{G}{2}\frac{D}{G}$ such combinations, 5) with **clustered RAID** two disks must fail

in the same cluster of size C (where $G \leq C$ is the parity group size), and there are $\binom{C}{2}\frac{D}{C}$ such combinations.

References

[1] Fault Tolerant Disk Drive Matrix, Patent 5,303,244, Granted April 12, 1994. *AT&T Global Information Solutions*.

[2] NonStop SQL, A Distributed, High-performance, High-reliablity Implementaion of SQL. Technical Report No. 82317, Tandem Database Group, March,1987.

[3] DBC/1012 database computer system manual release 2.0. Technical Report Document No. C10-0001-02, Teradata Corporation, Nov 1985.

[4] D. Bitton. Arm scheduling in shadowed disks. *COMPCON*, pages 132–136, Spring 1989.

[5] D. Bitton and J. Gray. Disk shadowing. *VLDB*, pages 331–338, 1988.

[6] P. Chen. An evaluation of redundant arrays of disks using an Amdahl 5890. Technical Report UCB/CSD 89/506, UC Berkeley, May 1989.

[7] P. Chen, G. A. Gibson, R. H. Katz, and D. A. Patterson. An evaluation of redundant arrays of disks using an Amdahl 5890. *ACM SIGMETRICS Conference*, pages 74–85, 1990.

[8] Peter M. Chen and David A. Patterson. Maximizing Performance in a Striped Disk Array. *ISCA*, pages 322–331, 1990.

[9] S. Chen and D. Towsley. Performance of a mirrored disk in a real-time transaction system. *ACM Sigmetrics 1991*, pages 198–207, 1991.

[10] S. Chen and D. Towsley. The Design and Evaluation of RAID5 and Parity Striping Disk Array Architecture. *Journal of Parallel and Distributed Computing*, pages 58–74, 1993.

[11] G. Copeland and T. Keller. A Comparison of High-Availability Media Recovery Techniques. *ACM SIGMOD Conference*, pages 98–109, 1989.

[12] David J. Dewitt, R. Gerber, G. Graefe, M. Heytens, K.Kumar, and M.Muralikrishna. Gamma : A high performance dataflow database machine. *VLDB Conference*, pages 228–240, 1986.

[13] S. Ghandeharizadeh and D. J. DeWitt. Hybrid-Range Partitioning Strategy: A New Declustering Strategy for Multiprocessor Database Machines. *VLDB*, pages 481–492, 1990.

[14] Garth A. Gibson. Performance and Reliability in Redundant Arrays of Inexpensive Disks. *1989 Computer Measurement Group (CMG) Annual Conference Proceedings*, December 1989.

[15] Leana Golubchik, John C.S. Lui, and Richard R. Muntz. Chained declustering: Load balancing and robustness to skew and failure. *RIDE-TQP Workshop*, February 1992.

[16] Jim Gray, Bob Horst, and Mark Walker. Parity striping of disk arrays: Low-cost reliable storage with acceptable throughput. *VLDB Conference*, pages 148–172, 1990.

[17] M. Holland and G. A. Gibson. Parity Declustring for Continuous Operation in Redundant Disk Arrays. In *5th Conference on Architectural Support for Programming Languages and Operating Systems*, Boston, MA, October 1992.

[18] M. Holland, G. A. Gibson, and D. P. Siewiorek. Architectures and Algorithms for On-Line Failure Recovery in Redundant Disk Arrays. *Submitted to the Journal of Distributed and Parallel Databases.*

[19] M. Holland, G. A. Gibson, and D. P. Siewiorek. Fast, On-Line Failure Recovery in Redundant Disk Arrays. In *23rd Annual International Symposium on Fault-Tolerant Computing*, 1993.

[20] H. Hsiao and D. J. DeWitt. Chained Declustering: A New Availability Strategy for Multiprocessor Database Machines. *Proc. of Data Engineering*, pages 456–465, 1990.

[21] R. Katz, D. W. Gordon, and J. A. Tuttle. Storage System Metrics for Evaluating Disk Array Organization.

[22] M. Y. Kim. Synchronized Disk Iterleaving. *IEEE Trans. on Computers*, pages 978–988, November 1986.

[23] M. S. Lakshmi and P. S. Yu. Effect of skew on join performance in parallel architectures. In *Int. Symposium on Databases in Parallel and Distributed Systems*, pages 107–120, 1988.

[24] E. Lee. Software and Performance Issues in the Implementation of a RAID Prototype. May 1990.

[25] E. Lee and R. Katz. Performance Consequences of Parity Placement in Disk Arrays. pages 190–199, 1991.

[26] J. Menon and J. Cortney. The Architecture of a Fault-Tolerant Cached RAID Controller. In *20th Annual International Symposium on Computer Architecture*, pages 76–86, San Diego, CA, May 1993.

[27] J. Menon and J. Kasson. Methods for Improved Update Performance of Disk Arrays. *Proceedings of the Hawaii International Conference on System Sciences*, pages 74–83, 1992.

[28] J. Menon and D. Mattson. Comparison of Sparing Alternatives for Disk Arrays. *Proceedings of the International Symposium on Computer Architecture*, 1992.

[29] A. Merchant and P. S. Yu. Design and Modeling of Clustered RAID. *Proceedings of the International Symposium on Fault-Tolerant Computing*, pages 140–149, 1992.

[30] Richard R. Muntz and John C.S. Lui. Performance analysis of disk arrays under failure. *VLDB Conference*, pages 162–173, 1990.

[31] David A. Patterson, Garth Gibson, and Randy H. Katz. A Case for Redundant Arrays of Inexpensive Disks (RAID). *ACM SIGMOD Conference*, pages 109–116, 1988.

[32] M. Schulze, G. Gibson, R. Katz, and D. Patterson. How-Reliable is a RAID? *COMPCON*, pages 118–123, 1989.

[33] D. Stodolsky, G. A. Gibson, and M. Holland. Parity Logging, Overcoming the Small Writes Problem in Redundant Disk Arrays. In *20th Annual International Symposium on Computer Architecture*, pages 64–75, San Diego, CA, May 1993.

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

[35] M. Stonebraker and G. A. Schloss. Distributed RAID - A New Multiple Copy Algorithm. *Sixth Int'l. Conf on Data Engineering*, pages 430–437, 1990.

[36] D. Towsley, S. Chen, and S. P. Yu. Performance analysis of a fault tolerant mirrored disk system. *Proceeding of Performance '90*, pages 239–253, 1990.

[37] Philip S. Yu and Asit Dan. Effect of system dynamics on coupling architectures for transaction processing. Technical Report RC 16606, IBM T.J. Watson Research Division, Feb 1991.

[38] Philip S. Yu and Asit Dan. Impact of affinity on the performance of coupling architectures for transaction processing. Technical Report RC 16431, IBM T.J. Watson Research Division, Jan 1991.

"Disk Cooling" in Parallel Disk Systems

Peter Scheuermann*
Dept. of Electr.. and Computer Sc.
Northwestern University
Evanston, IL 60208

Gerhard Weikum
Dept. of Computer Science
University of Saarbrucken
D-66041 Saarbrucken, Germany

Peter Zabback**
IBM Research Division
Almaden Research Center
650 Harry Road
San Jose, CA 95120

Abstract

Parallel disk systems provide opportunities for high performance I/O by supporting efficiently intra-request and inter-request parallelism. We review briefly the components of an intelligent file manager that performs striping on an individual file basis and achieves load balancing by judicious file allocation and dynamic redistribution of the data. The main part of the paper discusses our "disk cooling" procedure for dynamic redistribution of data which is based on reallocation of file fragments. We show that this heuristic method achieves excellent load balance in the presence of evolving access patterns. We report on two sets of experiments: a synthetic experiment which exhibits a self-similar skew in the data access patterns and a trace-based experiment where we study the impact of the file fragment size on the cooling procedure.

1 Introduction

Parallel disk systems are of great importance to massively parallel computers since they are scalable and they can ensure that I/O is not the limiting factor in achieving high speedup. However, to make effective use of commercially available architectures, it is necessary to develop intelligent software tools that allow automatic tuning of the parallel disk system to varying workloads. The choice of a striping unit and whether to choose a file-specific striping unit are important parameters that affect the response time and throughput of the system. Equally important are the decisions of how to allocate the files on the disks and how to perform redistribution of the files when access patterns change.

We have developed an intelligent file manager, called FIVE, for parallel disk systems that can perform striping on a file-specific or global basis, as desired by the application, and in addition it achieves load balancing by judicious file allocation and dynamic redistribution of data. Our system is geared toward software-controlled parallel disk systems in which each disk can be accessed individually. The system has the following salient properties:

- It consists of modular blocks that can be invoked independently; in particular the algorithms for file allocation and redistribution of data can be used regardless of whether striping is employed or not.

* The research of this author was partially supported by NASA-Ames grant NAG2-846 and by NSF grant IRI-9303583.
** This work was performed while the author was at ETH Zurich, Switzerland.

- It uses simple but effective heuristics that incur only little overhead.
- Its constituent algorithms can be invoked on-line, i.e., concurrently with regular requests to existing files.
- The heuristics for data placement and redistribution of data can be integrated with the fault tolerance techniques developed for RAIDs as well as various forms of data mirroring.

In this paper we discuss mainly our "disk cooling" procedure for dynamic redistribution of data which is based on reallocation of file fragments. We show that this heuristic method achieves excellent load balance in the presence of evolving access patterns. In addition, we also discuss opportunities for fine-tuning our disk cooling procedure so that the unit of reallocation can be chosen in order to account for the cost/benefit of the redistribution.

The remainder of this paper is organized as follows. In Section 2 we review briefly our file partitioning method and discuss the relationship between partitioning and load balancing. In Section 3 we present our load balancing procedure, concentrating on disk cooling and the bookkeeping steps required to keep track of its dynamically changing statistics. Section 4 reports on two sets of experiments: a synthetic experiment which exhibits a recursive skew of access and a trace-based experiment where we study the impact of the file fragment size on the cooling procedure. Section 5 concludes with a brief discussion of issues under investigation.

2 File Partitioning

File striping or declustering [10, 15] is a technique for file organization that divides a file into runs of logically consecutive data units, called "stripes" which are then spread across a number of disks in order to reduce the transfer time of a single request or to improve the throughput of multiple requests. The striping unit denotes the number of logically consecutive data bytes or blocks stored per disk, and the degree of declustering of a file denotes the number of disks over which a file is spread. In virtually all disk architectures that have been proposed so far, the striping unit is chosen globally [4]. This approach is suitable for scientific applications or pure on-line transaction processing, in which all files have approximately the same sort of access characteristics (i.e., only large requests or only single-block requests). However, as we have shown in [17] for many applications which exhibit highly diverse file access characteristics (e.g., VLSI design, desktop publishing, etc.) it is desirable to tune the striping unit individually for each file.

We have developed an analytic model for an open queueing system in order to determine heuristically the optimal striping unit on an individual file basis or on a global basis [17, 20, 22]. We observe here that an open queueing model is much more realistic for an environment where a large number of users issue requests to a parallel disk system, as compared to the closed queueing model used in [3, 12] where the number of concurrent I/O requests in the system is fixed. In our system, the striping unit is chosen in terms of data blocks. Our striping procedure can be applied to a file system in two different ways:

1. The striping unit of each file is chosen individually, based upon the file's average request size R. Two further options exist here. For low arrival rates of requests, where we can assume that no queueing delays occur, the response time can be computed as if the system operates in single user mode. For higher loads, the response time can be optimized by taking into account explicitly the arrival rate, λ, in addition to the parameter R.
2. The striping unit can be determined globally by any of the two options mentioned above based on the overall average request size $R^{\star}$.

Although the problems of file striping and load balancing are orthogonal issues, they are not completely independent. In order to derive analytically the optimal striping unit, it is assumed in [17, 20, 22] that the system is load balanced.

If the striping unit is a byte and all files are partitioned across all disks in the system, then we obtain a perfectly balanced I/O load. In general, very small striping units lead to a good load balancing. But throughput considerations require for many applications that we choose large striping units (e.g., the size of a cylinder). For example, the parity striping scheme proposed in [7] is based on very large (possibly infinite) striping units, and [14] proposes choosing both a small and a large striping unit for replicated data to support both on-line transaction processing and decision-support queries as well as batch processing. However, a coarser striping unit increases the probability of load imbalance under a skewed workload [13]. In general, we can see that file striping can help towards achieving good load balancing, but partitioning by itself is not sufficient for this goal. Additional methods for load balancing are called for, regardless of whether data partitioning is used or not.

3 Load Balancing

The load balancing component of our intelligent file system consists of two independent modules: one that performs file allocation [19] and the second that performs dynamic redistribution of data [17].

After the decision has been made to decluster a file over a particular number of disks, all striping units of the file that are to be mapped to the same disk are combined into a single allocation unit called an extent. The file allocation problem in a parallel disk system involves making a judicious decision about the disks on which to place the extents so as to optimize the load. While this problem is similar to the file allocation problem in distributed systems, it presents an additional constraint due to the need to consider also intra-request parallelism. This implies, that not all extents of a file should be allocated to the same disk if intra-request parallelism is to be supported.

In order to perform these load balancing procedures, i.e., file allocation and file redistribution, our file system keeps track of the following related statistics [5]:

- the heat of extents (or alternatively, of the smallest units of data migration) and disks, where the heat is defined as the number of block accesses of an extent or disk per time unit, as determined by statistical observation over a certain period of time,
- and the temperature of extents, which is defined as the ratio between heat and size.

3.1 File Allocation

A number of heuristic methods have been proposed for file allocation, with the simplest one being the round-robin scheme. A simple but effective heuristic algorithm for static file allocation, where all files are allocated at the same time and the heat of each extent can be estimated in advance, has been introduced in [5] (see, for example, [21] for a more sophisticated approach to statically load balanced data placement). The algorithm first sorts all extents by descending heat and the extents are allocated in sort order. For each extent allocation, the algorithm selects the disk with the lowest accumulated heat among the disks which have not yet been assigned another extent of the same file.

We have extended this greedy algorithm in order to deal with dynamic file allocation [19]. Since our algorithm makes no assumptions about the heat of a new file at allocation time, the sorting step is eliminated and the algorithm only uses the information about the heat of the files which have been allocated already and for which statistics are available. The disk selection can be made in such a way as to consider also, if so desired, the cost of additional I/Os necessary to perform partial disk reorganization. Partial disk reorganization may have to be performed if, due to file additions and deletions, there is room to store an extent on a disk but the space is not contiguous. Even more expensive is the situation when disk i has the lowest heat and may appear as the obvious choice to store a new extent of a file, but this disk does not have enough free space. In order to make room for the new extent we have to migrate one or more extents to a different disk. In order to account for these reorganization costs we associate with every disk a status variable with regard to the extent chosen for allocation. The status

Input: D - number of disks
H_j - heat of extent j
$H_i^\star$ - heat of disk i
$\overline{H}$ - average disk heat
E_i - list of extents on disk i sorted in descending temperature order
$\overline{D}$ - list of disks sorted in ascending heat order

Step 0: Initialization: target = not_found
Step 1: Select the hottest disk s
Step 2: Check trigger condition:
`if` $H_s > \overline{H} \times (1+\delta)$ `then`
Step 3: `while` (E_s not exhausted) and (target == not_found) `do`
Select next extent e in E_s
Step 4: `while` ($\overline{D}$ not exhausted) and (target == not_found) `do`
Select next disk t in $\overline{D}$ in ascending heat order
`if` (t does not hold an extent of the file to which e belongs)
`and` STATUS(t) == `FREE then`
target = found
`fi`
`endwhile`
`endwhile`
Step 5 : `if` s has no queue `then`
$H_s^{\star\prime} = H_s^\star - H_e$
$H_t^{\star\prime} = H_t^\star + H_e$
`if` $H_t^{\star\prime} < H_e$ `then`
reallocate extent e from disk s to disk t
update heat of disks s and t:
$H_s^\star = H_s^{\star\prime}$
$H_t^\star = H_t^{\star\prime}$
`fi`
`fi`
`fi`

Figure 1: Basic disk cooling algorithm

variable can take the values FREE, FRAG and FULL, depending upon whether the disk (1) has enough free space for the extent, (2) has enough space but the the space is fragmented or, (3) does not have enough free space. Our file allocation algorithm has the option of selecting disks in increasing heat order without regard to their status. Alternatively, we may select the disks in multiple passes, where in the first pass we only choose those that have status FREE.

3.2 The "Disk Cooling" Procedure

In order to perform dynamic heat redistribution we employ in our system a dynamic load balancing step, called disk cooling. Basically, disk cooling is a greedy procedure which tries to determine the best candidate, i.e., extent, to remove from the hottest disk in order to minimize the amount of data that is moved while obtaining the maximal gain. The temperature metric is used as the criterion for selecting the extents to be reallocated, because temperature reflects the benefit/cost ratio of the reallocation since benefit is proportional to heat (i.e., reduction of heat) and cost is proportional to size (of the reallocated extents). Our basic disk cooling procedure is illustrated in Figure 1. The extent to be moved, denoted by e, is reallocated on the coolest disk, denoted by t, such that t does not hold already an extent of the corresponding file and t has enough contiguous free space.

In our system the disk cooling procedure is implemented as a background demon which is invoked at fixed intervals in time. The procedure checks first if the trigger condition is satisfied or not (Steps 1 and 2 in Figure 1). If the trigger condition is false, the system is considered load balanced and no cooling action is performed. In the basic disk cooling procedure the system is not considered load balanced if the heat of the hottest disk exceeds the average disk heat by a certain quantity δ. It is important to observe that during each invocation of the procedure different disks can be selected as candidates for cooling after each cooling step.

Our procedure considers implicitly the cost/benefit ratio of a considered cooling action and only schedules it for execution if is considered beneficial. These cost considerations are reflected in Step 5 of the algorithm. The hottest disk is likely to have already a heavy share of the load, which we can "measure" by observing if its queue is non-empty. A cooling action would most likely increase the load imbalance if a queue is present at the source disk since it implies additional I/Os for the reorganization process. Hence, we choose not to schedule the cooling action if this condition is satisfied. We also consider the cooling move not to be cost-beneficial if, would it be executed, the heat of the target disk would exceed the heat of the source disk. Hence, although our background demon is invoked a fixed number of times, only a fraction of these invocations result in data migration.

Our generic disk cooling procedure can be generalized in a number of ways. In [16] we have shown how an explicit objective function based on disk heat variance (DHV) can be used in a more general test for the cost/benefit of a cooling action. Thus, the benefit is computed by comparing the DHV after the potential cooling step with the DHV before the potential cooling step. In addition, we can consider also explicitly the cost of performing the cooling. Thus, a more accurate calculation of benefit and cost would consider not only the reduction in heat on the origin disk and the increase in heat on the target disk, but also the additional heat caused by the reorganization process itself. The cooling process is executed during two intervals of time, the first corresponding to the read phase of the action and the second corresponding to the write phase of the action. The additional heat generated during these phases can be computed by dividing the size of the extent to be moved by the corresponding duration of the phase. The duration times of the read and write phase of a cooling action can be estimated by using a queueing model, as shown in [16].

Our disk cooling procedure can be fine-tuned so that the unit of reallocation is chosen dynamically in order to increase the potential of a positive cost/benefit ratio. In the basic procedure given in Figure 1 the unit of redistribution is assumed to be an extent. However, in the case of large extents that are very hot the cost of a redistribution may be prohibitive. In this case, we can subdivide further an extent into a number of fragments and use a fragment as the unit of redistribution. Since all fragments of an extent are of the same size we can now base the choice of the migration candidates (see Step 3 in Figure 1) on the heat statistic instead of temperature.

In addition, the increase in the number of allocation units of a file also requires that we remove the allocation constraint on the target disk, namely we do not require anymore that the disk should hold only one fragment of a file. Hence, we put here the objective of a balanced load above the requirement that the degree of declustering is optimal.

3.3 Heat Tracking

The dynamic tracking of the heat of blocks is implemented based on a moving average of the interarrival time of requests to the same block. Conceptually, we keep track of the times when the last k requests to each block occurred, where k is a fine-tuning parameter (in the range from 5 to 50). To illustrate this bookkeeping procedure, assume that a block is accessed at the points of time t_1, t_2,..., t_n ($n > k$). Then the average interarrival time of the k last requests is $\frac{t_n - t_{n-k+1}}{k}$, and the estimated heat of the block is the corresponding reciprocal $\frac{k}{t_n - t_{n-k+1}}$. Upon the next access to this block, say at time t_{n+1}, the block heat is re-estimated as $\frac{k}{t_{n+1} - t_{n-k+2}}$.

One may conceive an alternative method for heat tracking that keeps a count of the number of requests to a block within the last T seconds, where T would be a tuning parameter. The problem with such a request-count approach is that it cannot track the heat of both hot and cold blocks in an equally responsive manner. Hot blocks would need a relatively short value of T to ensure that we become aware of heat variations quickly enough. Cold

blocks, on the other hand, would need a large value of T to ensure that we see a sufficient number of requests to smooth out stochastic fluctuations. The moving-average method for the interarrival time does not have this problem since a fixed value of k actually implies a short observation time window for hot blocks and a long window for cold blocks. Moreover, extensive experimentation with traces from real applications with evolving access patterns has shown that our tracking method works well for a wide spectrum of k values; the heat estimation is fairly insensitive to the exact choice of k [22].

The adopted heat tracking method is very responsive to sudden increases of a block's heat; the new access frequency is fully reflected in the heat estimate after k requests, which would take only a short while for hot blocks (and reasonable values of k). However, the method adapts the heat estimate more slowly when a block exhibits a sudden drop of its heat. In the extreme case, a hot block may suddenly cease to be accessed at all. In this case, we would continue to keep the block's old heat estimate as there are no more new requests to the block. To counteract this form of erroneous heat estimation, we employ an additional "aging" method for the heat estimates. The aging is implemented by periodically invoking a demon process that simulates "pseudo requests" to all blocks. Whenever such a pseudo request would lead to a heat reduction, the block's heat estimate is updated; otherwise the pseudo request is ignored. For example, assume that there is a pseudo request at time t' and consider a block with heat H. We compute tentatively the new heat of the block as $\frac{H'=k}{t'-t_{n-k+2}}$, but we update the heat bookkeeping only if $H' < H$. The complete heat tracking method is illustrated in Figure 2.

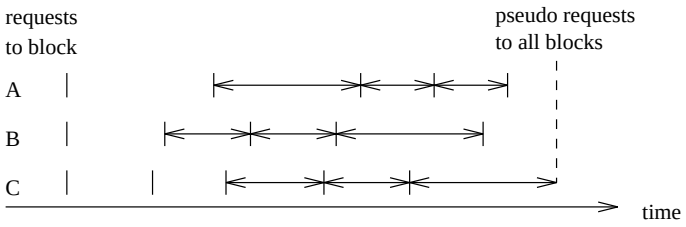


Figure 2: Illustration of the heat tracking method for $k = 3$. The relevant interarrival times are shown by the double-ended arrows.

The described heat tracking method requires a space overhead per block of $k + 1$ times the size of a floating-point number. Since we want to keep this bookkeeping information in memory for fast cooling decisions, it is typically unacceptable to track the heat of each individual block. For low-overhead heat tracking, we actually apply the heat estimation procedure to entire extents (or fragments of a specified size). We keep track of the times $t_n, \ldots, t_{n-k+1}$ of the last k requests that involve any blocks of the extent in the manner described above, and also we keep the number of accessed blocks within the extent for each of the last k requests. Assume that the average number of accessed blocks is R. Then the heat of the extent is estimated by $\frac{kR}{t_n - t_{n-k+1}}$. Finally, we estimate the heat of a fraction of an extent by assuming that each block in the extent has the same heat (which is extent heat divided by extent size). This extent-based heat tracking method provides a compromise between the accuracy and the space overhead of the block-based estimation procedure. The method has proven to be sufficiently accurate in all our experimental studies (including studies with application traces).

4 Performance Studies

In this section we present an experimental evaluation of our dynamic disk cooling procedure. In order to study the robustness of our procedure we performed two different sets of experiments. The first set of experiments was based on a synthetic workload, enabling us to study systematically the effect of changes in arrival rates and various patterns in data access skew. For the second set of experiments we used I/O traces from file servers at the University of California.

4.1 The Performance Testbed

Our testbed consists of a load generator, the file system prototype FIVE, and a simulated I/O system that is using CSIM [18]. FIVE allows for striping of files on an individual basis and incorporates heuristics for file striping, allocation, and dynamic load balancing. For the experiments presented here we wanted to study only the impact of the various parameters relevant to the disk cooling procedure. Hence, we performed striping on a global basis, using one track as the striping unit and static file allocation was performed using round-robin. The disk cooling procedure is invoked as a background process running concurrently with regular requests to existing files. FIVE can manage real data on real hardware or synthetic/real data on a simulated I/O system. For these experiments we used a simulated disk farm consisting of 32 disks whose parameters are described in Table 1. Note, that although average figures are given for the seek time and rotational latency, our detailed simulation computes the actual figures for each request by keeping track of the cylinder position of each disk arm.

capacity per disk [MBytes]	540	avg. seek time [ms]	12
# cylinder per disk	1435	avg. rotational latency [ms]	6.82
# tracks per cylinder	11	transfer rate [MBytes/s]	2.4
capacity per track [blocks]	35	block size [Bytes]	1024
# disks	32	total capacity [GBytes]	17

Table 1: Characteristics of the simulated disk farm

4.2 Synthetic Workload

We performed a number of synthetic experiments in order to study systematically the effects of different parameters on the disk cooling procedure. While experiments with real-life traces are important, it is often difficult to obtain long enough traces that exhibit all the relevant patterns of access. The purpose of the synthetic experiments was to study the impact of various arrival rates, degree of skew in data access, as well as fluctuations over time in the skew.

For these experiments we used $N = 1000$ files each having 70 blocks (2 tracks). Each file resided on two disks (i.e., the global striping unit was 1 track). Furthermore, as the file size was rather small, we considered only entire files as migration candidates and did not investigate smaller fragment sizes. Each (read or write) request accessed an entire file.[1] Note that this synthetic setup captures the most important workload features of applications such as office document filing, desktop publishing, etc., as far as load balancing issues are concerned. In a real application, there would probably be more files and also larger files, but, on the other hand, I/O requests would often access only a fraction of a file and a large fraction of the files would be accessed only extremely infrequently. So we disregard the non-essential details for the sake of simplifying the experiments.

The I/O requests were generated so as to exhibit a self-similar skew in the data access pattern across the files [8]. We use an X/Y distribution of access, where X % of the requests are addressed to Y % of the files. Thus, if the files are numbered from 1 to N, for a given setting of the parameters X and Y, the probability of accessing a file numbered i, with $i \leq s$, is given by the formula of [11]:

$$Prob(i \leq s) = \left(\frac{s}{N}\right)^{\log(X/100)/\log(Y/100)}$$

In order to experiment with fluctuations in patterns of access, we have implemented a uniform shift procedure which allows us to switch among the files the heats assigned to them from one period to the other. Let us assume that during simulation period i the files numbered 1 through N have been assigned heats in decreasing order of

[1] In Section 4.3 we discuss experiments with larger file sizes and variable request sizes.

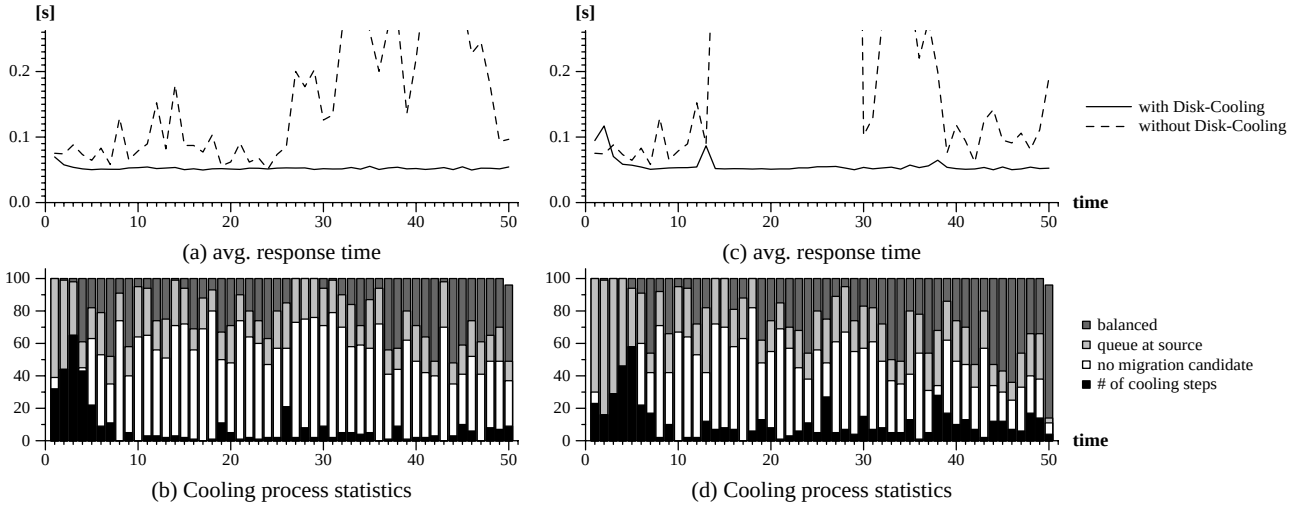


Figure 3: Average response time for synthetic experiments. skew 70/30. $\lambda = 200$. (a) and (b) one shift, (c) and (d) three shifts.

magnitude; thus file numbered 1 was assigned the highest heat, h_1, and so on, with the file numbered N being assigned the smallest heat, h_N. A shift in heat of magnitude r means that in the next simulated period, namely $i + 1$, the highest heat h_1 is assigned to file numbered $1 + r$, h_2 is assigned to file numbered $2 + r$, and so on in cyclic fashion.

In all experiments the simulation time was divided into 50 intervals. We report here on two sets of experiments using a different degree of skew: the first one uses a 80/20 skew in access, while the second one uses a 70/30 skew. For each set of experiments we experimented with different shifts in skew: no shift for the entire simulation versus one or three shifts in skew. In the case of one shift the magnitude chosen was $r = 500$ and the shift occurred in the middle interval (number 25). In the experiments with three shifts the magnitude of the shift was 250 and the shifts occurred during intervals 12, 25 and 37, respectively. Different arrival rates were used for each set of experiments, but due to lack of space we limit ourselves here to only one arrival rate per X/Y skew. Note that a lower skew in data access can handle a much higher arrival rate. This is due to the fact that for high degrees of skew the vanilla round-robin allocation method thrashes above a certain arrival rate.

Figure 3 depicts the average response times and the cooling frequencies as they vary over the simulated time period for a skew of 70/30. Figure 4 repeats these measurements for a skew of 80/20. The disk cooling procedure achieves tremendous savings in the response time due to better load balancing, and hence reduced queueing delays.

The histograms illustrated in the Figures depict the frequency of the data migration steps invoked by our cooling procedure, varying over the simulation intervals. The disk cooling procedure is implemented as a background demon which is invoked a fixed number of times (i.e., 100 times) during each simulation period. The histograms illustrate how many of these invocations actually resulted in data migrations being executed (cooling steps). An invocation will not result in a cooling action if the system determines that the cost/benefit ratio is not favorable. The cases when no cooling actions occur are divided into two categories in our histograms: *queue at source* and *no migration candidates*. The *queue at source* category counts those invocations where a queue is observed at the source disk. The *no migration candidate* category includes those invocations where (1) all extents on the hottest disk are so hot that after a move the target disk would become hotter than the source disk before the move, or (2) all remaining extents have no observed I/Os. The first case was discussed already in Section 3. The second case is related to the fact that our disk cooling procedure has no a-priori knowledge about the heat of any of the extents. Hence, initially the heat of any extent is assumed to be zero and the disk cooling procedure is not executed further when we reach extents with no observed I/Os.

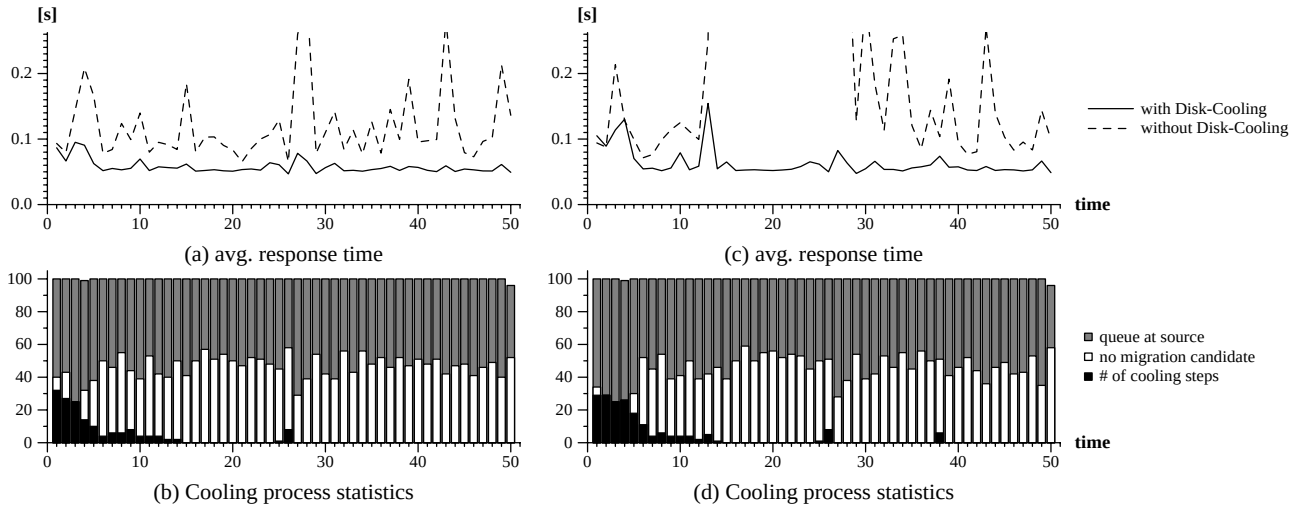


Figure 4: Average response time for synthetic experiments. Skew 80/20. $\lambda = 95$. (a) and (b) one shift, (c) and (d) three shifts.

In all experiments our algorithm initiated a larger number of cooling steps during the initial learning phase. Figure 4 shows that in the case of a 80/20 skew, the number of cooling steps initiated subsequently was rather small. In addition, the experiments illustrate that our disk cooling procedure reacts fast to changes in the access pattern. As observed in Figure 4, a sudden shift during interval 25 caused a slight increase in the average response time. But the system is fast to recognize it, and immediately in the next time interval cooling actions are taken to readjust the load balance. The experiments with three shifts show a similar pattern: very few and singular cooling actions, performed after the shifts occurs. On the other hand, the experiments depicted in Figure 3 show a different configuration. In these experiments we observe that the cooling steps occur continuously throughout the entire simulation period. This is due to the fact that for a more moderate skew the disk cooling procedure has more degrees of freedom for the migration. With a high skew it often happens that we end up with one or two disks which have only one extent, namely the hottest ones. When this occurs, no suitable target disk can be found for the migration since its heat will now become the bottleneck.

4.3 Trace-based Workload

For this experimental study we used a 48-hours trace from the University of California, Berkeley, a period during which the majority of the applications dealt with VLSI design and simulation. The original trace, described in [1], was reformatted in order to feed it into our testbed. Furthermore, we removed all short-lived files with a lifetime of less than a minute, assuming that these files would be cached on a permanent basis. An important feature of this trace is the constantly changing pattern of access to the individual files. Files accessed frequently at the beginning of the trace are hardly requested anymore at the end of the trace. Hence, we are dealing here with a skewed access distribution that is undergoing continual shifts. In addition, the file sizes are substantially larger than in the synthetic experiment. This enabled us to study here the impact of the file fragment size on the disk cooling procedure.

The trace consists of approximately 440,000 requests for data in 14,800 files. The average request size is about 107 KBytes, but a wide variance in the request sizes is exhibited. The original average arrival rate of the trace was $\lambda = 2.45$ requests per second; we accelerated the load by a "speed-up factor" of 10 for a stress test, thus the effective average arrival rate was $\lambda = 25$ requests per second. Note, however, that the trace contained heavy load surges with much higher arrival rates. All files were striped with a striping unit of one cylinder (385 blocks) and allocated on the disks in a round-robin manner.

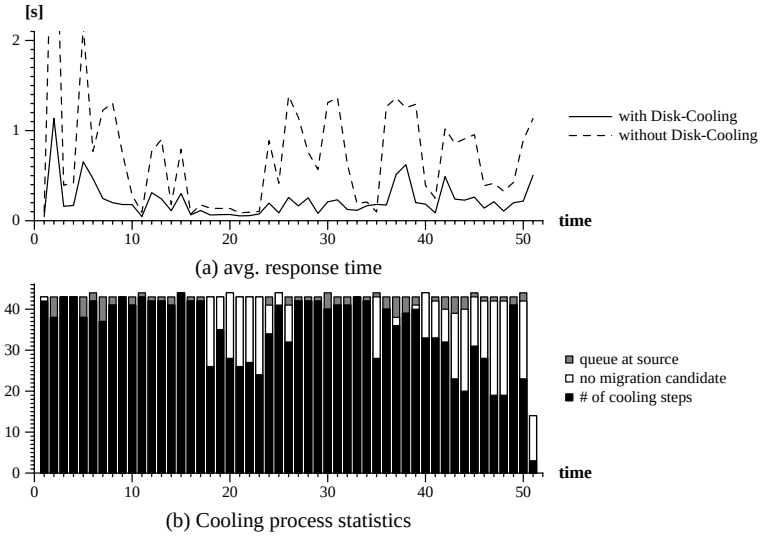


Figure 5: Response time for the trace-driven experiment

As in the synthetic experiments, the disk cooling procedure had no a-priori information about the access frequencies of the individual extents or fragments. The heat statistics were collected and updated dynamically using a moving window containing the last $k = 20$ requests to a given extent (fragment).

In our first experiment we assumed that the fragment size for the reallocation procedure is 210 Kbytes (six tracks). Figure 5 (a) shows the average response time varying over time obtained by using our disk cooling procedure versus a vanilla method that does not make use of disk cooling. As expected, our load balancing algorithm exhibits a learning phase at the beginning of the simulation, during which it collects heat statistics, which corresponds to the peak in access time of Figure 5 (a). After this the average response time obtained with disk cooling drops substantially. Over the entire simulation period the average response time measured was 1.085 seconds without disk cooling, versus 0.297 seconds with disk cooling.

Figure 5 (b) illustrates the frequency of the data migration steps invoked by our load balancing procedure. We observe that the cooling steps are executed throughout the entire simulation period due to the continually evolving pattern of access, i.e., the constant shift in skew. Overall, 1798 cooling steps were executed each requiring on the average 0.13 seconds. Between simulation periods 17 and 28, and then between periods 40 and 49, the cooling quiescents somewhat. This is due to the fact that the trace covers a 48 hours period and these periods correspond to the two lightly loaded night intervals (as backup activity was not recorded in the trace).

In order to study the impact of file fragment size on the disk cooling procedure we designed a further set of experiments. The file fragment size was varied across experiments from FG=0, (no fragmentation — the extent is moved in its entirety) to FG =35 KBytes (one track). For each experiment, i.e., simulation period, the fragment size was kept fixed. Figure 6 shows the average response times with disk cooling for different fragment sizes. We observe that in the extreme case where no fragmentation is used (FG=0) the disk cooling procedure performs almost identically to the vanilla round-robin allocation method (see Figure 5 (a) for comparison). This phenomenon is due to two factors characteristic to large files. First, the extents can become so large that the cost of the migration exceeds the benefit of the move. Secondly, the number of extents in a file can be quite large, hence it becomes difficult to find a target disk which satisfies the constraint that no other extents of the file are already stored there. At the other extreme setting of the fragment size, i.e., FG = 35 Kbytes (one track) we also observed no improvement versus the vanilla algorithm which does not perform disk cooling, since the benefits of each migration are too small. However, for fragment sizes of 140 Kbytes, 210 Kbytes or 385 Kbytes (one cylinder) we observed a substantial improvement of the response time. All these fragment sizes offer a good compromise between costs and benefit of the redistribution. Furthermore, it is worthwhile to observe that the

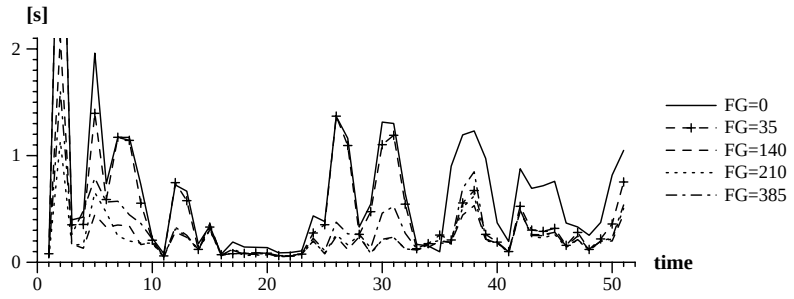


Figure 6: Response time for different fragment sizes

response times are fairly insensitive to the exact settings of the fragment size.

The trace-based experiments confirm the results obtained in the synthetic experiments. The cooling procedure is robust and fine-tuned with respect to changes in access patterns. If the pattern of access changes at certain time intervals and remains relatively stable in between, as was the case in the 1 shift/ 3 shifts experiments, the load balance can be restored with very few cooling steps if the skew is large. On the other hand, for smaller values of data skew, or for a constant shift in skew, as happened with the trace, the migration activities occur with high frequency. In all cases, the migration is executed only if the benefits of cooling exceed the cost of the move and this trade-off presents opportunities for fine-tuning the choice of the migration unit (i.e., the fragment size).

5 Further Issues

The disk cooling method that we have discussed in detail in this paper forms a major component in an integrated but modular set of data placement and dynamic reorganization heuristics that we have developed for parallel disk systems. This package is implemented in the FIVE experimental file system. We are in the process of investigating additional dimensions of the cooling method and exploring various generalizations of our approach.

Most importantly, we are considering the impact of caching on our heat tracking method. For this purpose, we distinguish the logical heat and the physical heat of a block, where the former includes all accesses to a block and the latter counts only those accesses that incurred disk I/O. While it may seem at a first glance that disk load balancing needs to consider only physical heat, such an approach would be very sensitive to sudden changes in the cache effectivity. For example, sudden load imbalances may result from a decision of the cache manager to drop several hot blocks that happen to reside on the same disk. This could occur when the cache manager needs to set aside a large amount of working memory (e.g., for a hash join), which effectively changes the memory size as far as disk block caching is concerned. On the other hand, simply balancing the logical heat of the data does not necessarily achieve the goal of disk load balancing, since different disks may have different quantities of “cached heat” (i.e., accesses that are serviced by the cache).

Another important issue that needs further investigation is the synchronization and fault tolerance of the various reorganization steps that we may invoke on-line in our system (e.g., the cooling steps). Basically, this requires critical sections for accesses to the block addressing tables and also logging the changes to these tables. We are currently working out details of these issues. Note that none of the on-line reorganizations requires holding locks on the actual data blocks for transaction-length or longer duration.

Finally, we are aiming to generalize our approaches to data partitioning, data allocation, and dynamic load balancing to arbitrary distributed storage systems such as shared-nothing parallel computers or workstation farms. While basic principles of our approach can be carried over, differences in the various cost metrics demand us to reconsider the benefit/cost optimization. We have also started studying a broader spectrum of application workloads. In particular, we want to address the performance requirements of multimedia applications with accesses to continuous, delay-sensitive media like audio and video. Although there exists some promising work on this

subject (e.g., [2, 6, 9]), we believe that substantially more research is needed towards efficient and cost-effective multimedia data management.

References

[1] Baker, M.G., Hartman, J.H., Kupfer, M.D., Shirriff, K.W., and Ousterhout, J.K., "Measurements of a Distributed File System," Proc. 13th ACM Symposium on Operating Systems Principles, 1991.

[2] Chiueh, T., and Katz, R., "Multi-Resolution Video Representation for Parallel Disk Arrays," Proc. ACM Multimedia Conf., 1993.

[3] Chen, P.M. and Patterson, D.A.," Maximizing Performance in a Striped Disk-Array," Proc. 17th Int. Symposium on Computer Architecture, 1990.

[4] Chen, P.M., Lee, E.K., Gibson, G.A., Katz, R.H., and Patterson, D.A., "RAID: High-Performance, Reliable Secondary Storage," Technical Report UCB/CSD-93-778, Department of Computer Science, University of California at Berkeley, 1993.

[5] Copeland, G., Alexander, W., Boughter, E., and Keller, T., "Data Placement in Bubba," Proc. ACM SIGMOD Conf., 1988.

[6] Gemmel, J., and Christodoulakis, S., "Principles of Delay-Sensitive Multimedia Data Storage and Retrieval," ACM Transactions on Information Systems, Vol. 10, No. 1, 1992.

[7] Gray, J.N., Horst B., and Walker, M., "Parity Striping of Disc Arrays: Low-Cost Reliable Storage with Acceptable Throughput," Proc. 16th Int. Conf. on Very Large Data Bases, 1990.

[8] Gray, J., Sundaresan, P., Englert, S., Baclawski, K., and Weinberger, P.J., "Quickly Generating Billion-Record Synthetic Databases," Proc. ACM SIGMOD Int. Conf., 1994.

[9] Ghandeharizadeh, S., and Ramos, L., "Continuous Retrieval of Multimedia Data Using Parallelism," IEEE Transactions on Knowledge and Data Engineering, Vol. 5, No. 4, 1993.

[10] Kim, M.Y.,"Synchronized Disk Interleaving," IEEE Transactions on Computers, Vol. C-35, No. 11, 1986.

[11] Knuth, D.E.,"The Art of Computer Programming. Vol. 3: Sorting and Searching," Addison-Wesley, 1973.

[12] Lee, E.K., and Katz, R.H., "An Analytic Performance Model of Disk Arrays," Proc. ACM SIGMETRICS Conf., 1993.

[13] Livny, M., Khoshafian, S., and Boral, H., "Multi-Disk Management Algorithms," Proc. ACM SIGMETRICS Conf., 1987.

[14] Merchant, A. and Yu, P.S.,"Performance Analysis of a Dual Striping Strategy for Replicated Disk Arrays," Proc. 2nd Int. Conf. on Parallel and Distributed Information Systems, 1993.

[15] Salem, K., and Garcia-Molina, H.,"Disk Striping," Proc. 2nd Int. Conf. on Data Engineering, 1986.

[16] Scheuermann, P., Weikum, G., and Zabback, P., "Adaptive Load Balancing in Disk Arrays," Proc. 4th Int. Conf. on Foundations of Data Organization and Algorithms, Lecture Notes in Computer Science, No. 730, 1993.

[17] Scheuermann, P., Weikum, G., and Zabback, P., "Data Partitioning and Load Balancing in Parallel Disk Systems," Technical Report 209, Department of Computer Science, ETH Zurich, January 1994.

[18] Schwetman, H., "CSIM Reference Manual (Revision 16)," Technical Report ACA-ST-252-87, MCC, 1992.

[19] Weikum, G., Zabback, P., and Scheuermann, P., "Dynamic File Allocation in Disk Arrays," Proc. ACM SIGMOD Int. Conf., 1991.

[20] Weikum, G., and Zabback, P., "Tuning of Striping Units in Disk-Array-Based File Systems," Proc. 2nd Int. Workshop on Research Issues on Data Engineering: Transaction and Query Processing (RIDE-TQP), 1992.

[21] Wolf, J., "The Placement Optimization Program: A Practical Solution to the Disk File Assignment Problem," Proc. ACM SIGMETRICS Conf., 1989

[22] Zabback, P., "I/O Parallelism in Database Systems," Ph.D. Thesis (in German), Department of Computer Science, ETH Zurich, 1994.

Issues in Parallel Information Retrieval

Anthony Tomasic Hector Garcia-Molina
Stanford University, Stanford, CA 94305-2140
email:{tomasic,hector}@cs.stanford.edu

Abstract

The proliferation of the world's "information highways" has renewed interest in efficient document indexing techniques. In this article, we provide an overview of the issues in parallel information retrieval. To illustrate, we discuss an example of physical index design issues for inverted indexes, a common form of document index. Advantages and disadvantages for query processing are discussed. Finally, to provide an overview of design issues for distributed architectures, we discuss the parameters involved in the design of a system and rank them in terms of their influence on query response time.

1 Introduction

As the data volume and query processing loads increase, companies that provide information retrieval services are turning to parallel storage and searching. The idea is to partition large document collections, and their index structures across computers. This allows for larger storage capacities and permits searches to be in parallel.

In this article we sample research in the area of parallel information retrieval. We start by summarizing basic information retrieval concepts, and then describe how they have been applied in a parallel environment. We also give a short summary of our own research in this area, mainly as an example of the types of algorithms that need to be developed, and the system issues that need to be studied.

2 Information Retrieval Basics

For an introduction to full-text document retrieval and information retrieval systems, see reference [16]. An *information retrieval model* (IRM) defines the interaction between a user and an information retrieval system and consists of three parts: a *document representation*, a *user need* and a *matching function*.

The boolean IRM is provided by most existing commercial information retrieval systems. Its document representation is the set of words that appear in each document. Typically, each word is also *typed* to indicate if it appears in the title, abstract, or some other field of the document. The boolean IRM user need is represented by a boolean *query*. A query consists of a collection of pairs of words and types structured with boolean operators. For example the query *title information and title retrieval or abstract inverted* contains three pairs and two operators. The matching function of a query in the boolean IRM is boolean satisfiability of a document representation with respect to the query.

The vector IRM is popular in academic prototypes for information retrieval systems and has recently gained commercial acceptance. Its document representation is the set of words that appear in each document and an associated *weight* with each word. The weight indicates the "relevance" of the corresponding word to the document. Thus, a document is represented as a vector. A vector IRM user need is represented by another vector (this vector can be extracted from a document or a set of words provided by a user). The matching function computes the similarity between the user need and the documents. Thus, all the documents can be ranked with respect to

the similarity. Typically, the topmost similar documents are returned to the user as an answer. There is much research on the assignment of weights to words and on the effectiveness of various matching functions for information retrieval. However, both the boolean IRM and the vector IRM and associated variation of these models can be computed efficiently with inverted lists. (See Section 4 for a description of inverted lists.) Reference [28] surveys information retrieval models.

The focus of traditional information retrieval research is to develop IRMs that provide the most *effective* interaction with the user. Our focus in this article, however, is in providing the most *efficient* interaction with the user in terms of response time, throughput and other measures, regardless of which IRM is used.

In the design of full-text document retrieval systems, there is a basic trade-off between the time to process the document database and the time to process queries. Broadly speaking, the more time spent processing the document database (i.e., building indexes) the less time is spent processing queries. In some scenarios (such as government monitoring of communication), a tremendous amount of information must be queried by only a few queries. In this case, time spent indexing is wasted and linear searching of documents is more efficient. Work in this area concentrates on hardware processors for speeding up the scanning of text [11]. More typically, indexing the documents is worthwhile because the cost can be amortized across many queries. We consider only these latter systems.

Emrath's thesis [6] explores this trade-off between query and update time by providing a data structure that can be tuned in the amount of information indexed. Essentially, the database is partitioned into equal sized "pages." A page is a fixed number of words located together in a document. Duplicate occurrences of words are dropped within a page. If the page is large, many duplicates are dropped from the index, speeding up indexing time. If the page is small, few duplicate words are dropped, slowing down indexing time. For certain applications this tuning of the data structure works well.

More recent work [18, 26, 27] uses physical index design to express the trade-off. The collection of documents is partitioned and each partition has an independent index at the physical index design level, but the entire collection has a single logical index. This provides fast update time but slow query time since each physical index must be searched. To provide fast query time, the physical indexes are merged according to a family of algorithms. More typically, indexing the documents in a single physical index is worthwhile because the cost can be amortized across many queries. We consider only these latter systems for the remainder of this article.

Much research has gone into designing data structures for indexing text. Faloutsos [7] is a survey of this issue. One approach is the use of *signature schemes* (also known as superimposed coding) [13]. Here, each word is assigned a random (hashed) k-bit code of an n-bit vector – for example the word "information" might correspond to bit positions 10 and 20 of a 2 kilobyte vector. Each document is represented by an n-bit vector containing the union of all the k-bit codes of all the words in the document. Queries are constructed by producing an n-bit vector of the k-bit codes of the words in the query. Matching is performed by comparing a query against the document vectors in the database. This scheme is used because the signatures of documents can be constructed in linear time. Unfortunately, the matching process produces "false drops" where different words or combinations of words are mapped into the same k-bit codes. One approach is to ignore false drops and inform the user that some additional documents may be returned. We do not consider this approach further. Otherwise, each document in the result of the matching process must be checked for false drops. While the number of false drops can be statistically controlled for the average case, the worst-case behavior of this data structure implies checking *every* document in the database for some queries, which is prohibitively expensive for large document collections. Lin [14] describes a signature scheme where multiple independent signatures are used to control false drops and to improve parallel performance.

Another data structure is PATRICIA trees and PAT arrays [9, 10]. Here, the database is represented as one *database string* by placing documents end-to-end. A tree is constructed that indexes the semi-infinite strings of the database string. A semi-infinite string is a substring of the database string starting at some point and proceeding until it is a unique substring. The PAT system provides indexing and querying over semi-infinite strings. The New Oxford English Dictionary has be index using this data structure. The query time, indexing time, and

storage efficiency are approximately the same as inverted lists. The techniques described here can be applied to this data structure.

For commercial full-text retrieval systems, inverted files or inverted indexes [8, 13] are typically used. Note that the information represented in each posting (each element of an inverted list) varies depending on the type of information retrieval system. For a boolean IRM full-text information retrieval system, the posting contains the document identifier and the position (as a byte offset or word offset from the beginning of the document) of the corresponding word. For a boolean IRM abstracts text information retrieval system, the posting contains the document identifier without a positional offset (since duplicate occurrences of a word in a document are not represented in these systems). For a vector IRM full-text or abstracts information retrieval systems the posting contains the document identifier and a weight. All of the above systems can be typed. In this case, the type system can be encoded by setting aside extra bits in each posting to indicate which fields the word appears in the document. Other methods of representing the type information are also used. As the information retrieval model becomes more complicated, more information is typically placed in each posting.

A related area of research is the compression of inverted indexes [29, 30]. The inverted index for a full-text information retrieval system is very large – typically on the same scale in size as the text. In fact, the original documents (minus punctuation) can be reconstructed from the inverted index. Thus, one interesting physical design issue is the impact of the compression ratio of the inverted index on response time. We return to this issue in Section 6.

3 Parallel Query Processing

Various distributed and parallel hardware architectures can be applied to the problem of information retrieval. A series of papers by Stanfill studies this problem for a Connection Machine. In reference [20], signature schemes are used. A companion paper by Stone [22] argues that inverted lists on a single processor are more efficient. In reference [21], inverted lists are used to support parallel query processing (in a fashion similar to that used by the system index organization that will discussed in Section 4). Finally, in reference [19], an improvement of the previous paper based on the physical organization of inverted lists is described. The technique essentially improves the alignment of processors to data.

An implementation of vector IRM full-text information retrieval is described in reference [1] for the POOMA machine. The POOMA machine is a 100-node, 2-d mesh communication network where each node has 16 MB of memory and a processor. One out of five nodes has an ethernet connection and one half of the nodes have a local disk. The implementation partitions the documents among the processors and builds a local inverted index of the partition. (This approach is similar to the host index organization of Section 4; however there are two processors per disk, as opposed to multiple disks per processor.) This paper cites a 2.098 second estimated query response time for a 191-term query on a database of 136,020 documents with a 20-node machine.

Some preliminary experimental results are reported in reference [3] for a 16 processor farm (Meiko Computing Surface). The vector IRM is used here and a signature scheme is used as the data structure. Unfortunately, the database has only 6,004 documents and the query workload only 35 queries.

The performance of some aspects of query and update processing of an implementation of a boolean IRM full-text information retrieval is discussed in reference [5] for a symmetric shared-memory multiprocessor (Sequent).

Reference [15] presents a discussion of the architecture issues in implementing the IBM STAIRS information retrieval system on a network of personal computers. This paper argues for the physical distribution of inverted lists across multiple machines when the size of a single database is larger than the storage capacity of a node on the network. This idea is essentially a special case of disk striping, where an object (in this case an inverted list) is partitioned across disks.

In the analysis of query processing, a query can be divided into three parts: parsing the query, matching the query against the database, and retrieving the documents in the answer. Parsing consumes few resources and is

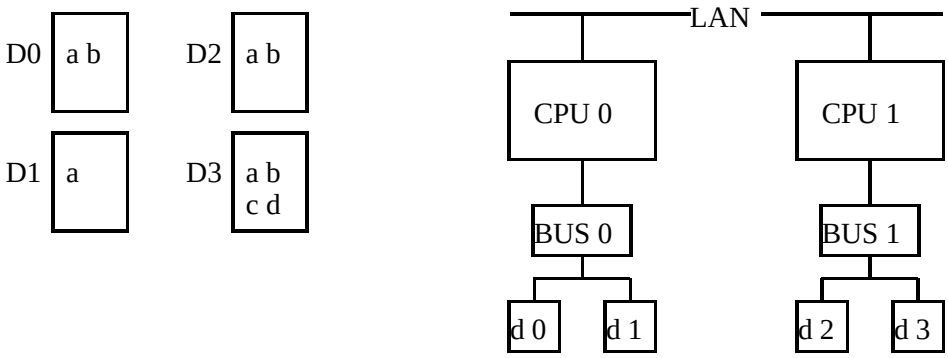


Figure 1: A example set of four documents and an example hardware configuration.

typically the same for all information retrieval systems. Retrieving of documents offers some interesting issues (such as placement of the documents) but again few resources are needed. Burkowski [2] examines the performance problem of the interaction between query processing and document retrieval and studies the issue of the physical organization of documents and indices. His paper models queries and documents analytically and simulates a collection of servers on a local-area network.

Schatz [17] describes the implementation of a distributed information retrieval system. Here, performance improvements come from changing the behavior of the interface to reduce network traffic between the client interface and the backend information retrieval system. These ideas are complementary our work. Three improvements are offered. First, summaries of documents (or the first page) are retrieved instead of entire documents. This scheme reduces the amount of network traffic to answer an initial query and shortens the time to present the first result of a query, but lengthens the time to present the entire answer. Second, "related" information such as document structure definitions are cached to speed up user navigation through a set of documents. Third, the contents of documents (as opposed to summaries) are prefetched while the user interface is idle.

Our own work [23, 24, 25] compares various options for partitioning an inverted list index across a shared-nothing multi-processor. (Reference [12] considers shared-everything multi-processors.) Simulated query loads are used in [24, 25], while [23] uses a trace-driven simulation.

4 Some Physical Design Choices

To illustrate more concretely the types of choices that are faced in partitioning index structures across machines, in this section we briefly describe the choices for an inverted-lists index, using the terminology of [25]. As stated earlier, this is the most popular type of index in commercial systems.

The left hand side of Figure 1 shows four sample documents, D0, D1, D2, D3, that could be stored in an information retrieval system. Each document contains a set of words (the text), and each of these words (maybe with a few exceptions) are used to index the document. In Figure 1, the words in our documents are shown within the document box, e.g., document D0 contains words a and b.

As discussed in Section 1, full-text document retrieval systems traditionally build inverted lists on disk to find documents quickly [8, 13]. For example, the inverted list for word b would be b: (D0,1), (D2,1), (D3,1). Each pair in the list is a posting that indicates an occurrence of the word (document id, position). To find documents containing word b, the system needs to retrieve only this list. To find documents containing both a and b, the system could retrieve the lists for a and b and intersect them. The position information in the list is used to answer queries involving distances, e.g., find documents where a and b occur within so many positions of each other.

Index	Disk	Inverted Lists in *word: (Document, Offset)* form
Host	d 0	a: (D0, 0), (D1, 0)
	d 1	b: (D0, 1)
	d 2	a: (D2, 0), (D3, 0); c: (D3, 2)
	d 3	b: (D2, 1), (D3, 1); d: (D3, 3)
System	d 0	a: (D0, 0), (D1, 0), (D2, 0), (D3, 0)
	d 1	b: (D0, 1), (D2, 1), (D3, 1)
	d 2	c: (D3, 2)
	d 3	d: (D3, 3)

Table 2: The various inverted index organizations for Figure 1.

Suppose that we wish to store the inverted lists on a multiprocessor like the one shown on the right in Figure 1. This system has two processors (CPUs), each with a disk controller and I/O bus. (Each CPU has its own local memory.) Each bus has two disks on it. The CPUs are connected by a local area network. Table 2 shows four options for storing the lists. The host and I/O bus organizations are identical in this example because each CPU has only one I/O bus.

In the *system* index organization, the full lists are spread evenly across all the disks in the system. For example, the inverted list of word b discussed above happened to be placed on disk $d1$. This organization essentially divides the keywords among the processors.

In the *host* index organization, documents are partitioned into two groups, one for each CPU. Here we assume that documents D0, D1 are assigned to CPU 0, and D2, D3 to CPU 1. Within each partition we again build inverted lists. The lists are then uniformly dispersed among the disks attached to the CPUs. For example, for CPU 1, the list for a is on $d2$, the list for b is on $d3$, and so on.

Clearly, many choices are available for physical index organization beyond those described here. We cannot consider all possible organizations. Our criteria for choosing these two organizations focuses first on the optimization of queries as opposed to updates. Thus, we assume that the inverted lists on each machine are stored contiguously on disk. Second, we are interested in the interaction between the physical index organization and the allocation of resources (CPUs, disks, I/O buses) of a shared-nothing distributed system. In addition, we have studied issues such as striping and caching of the physical index organization with respect to a single host.

5 Query Processing

Given a physical index partition like the ones illustrated in the previous section, how does one process queries? To illustrate, let us focus on a particular type of query, a "boolean and" query. Such queries are of the form $a \wedge b \wedge c \ldots$, and find the documents containing all the listed words. The words appearing in a query are termed *keywords*. Given a query $a \wedge b \ldots$ the document retrieval system generates the *answer set* for the document identifiers of all the documents that match the query. A *match* is a document that contains the words appearing in the query.

Notice that boolean-and queries are the most primitive ones. For instance, a more complex search such as $(a \wedge b)$ OR $(c \wedge d)$ can be modeled as two simple and-queries whose answer sets are merged. A distance query "Find *a* and *b* occurring within *x* positions" can be modeled by the query $a \wedge b$ followed by comparing the positions of the occurrences. Thus, the query processing strategies for the more complex queries can be based on the strategies we will illustrate here for the simple boolean-and queries.

For the host index organization, boolean-and queries can processed as follows. The query $a \wedge b \ldots$ is initially processed at a *home* site. That site issues *subqueries* to all hosts; each subquery contains the same keywords as

the original query. A subquery is processed by a host by reading all the lists involved, intersecting them, and producing a list of matching documents. The answer set of a subquery, termed the *partial answer set*, is sent to the home host, which concatenates all the partial answer sets to produce the answer set.

In the system index organization, the subquery sent to a given host contains only the keywords that are handled by that host. If a host receives a query with a single keyword, it fetches the corresponding inverted list and returns it to the home host. If the subquery contains multiple keywords, the host intersects the corresponding lists, and sends the result as the partial answer set. The home host intersects (instead of concatenates) the partial answer sets to obtain the final answer.

There are many interesting trade-offs among the storage organizations and query processing strategies. For instance, with the system index organization, there are fewer I/Os. That is, the a list is stored in a single place on disk. To read it, the CPU can initiate a single I/O, the disk head moves to the location, and the list is read. (This may involve the transfer of multiple blocks). In the host index organization, on the other hand, the a list is actually stored on, say, 4 processors. To read these list fragments, 4 I/Os must be initiated, four heads must move, and four transfers occur. However, each of the transfers is roughly a fourth of the size, and they can take place in parallel. So, even though we are consuming more resources (more CPU cycles to start more I/Os, and more disk seeks), the list may be read more quickly.

The system index organization may save disk resources, but it consumes more resources at the network level. Notice that in our example, the entire *c* list is transferred from CPU 1 to CPU 0, and these inverted lists are usually much longer than the document lists exchanged under the other schemes. However, the long inverted list transfers do not occur in all cases. For example, the query "Find documents with a and b" (system index organization) does not involve any such transfers since all lists involved are within one computer. Also, it is possible to reduce the size of the transmitted inverted lists by moving the shortest list. For example, in our "Find documents with a and c", we can move the shorter list of a and c to the other computer.

It is also important to notice that the query algorithms we have discussed can be optimized in a variety of ways. To illustrate, let us describe one possible optimization for the system index organization. We call this optimization *Prefetch I*; it is a heuristic and in some cases it may not actually improve performance. (Other query optimization techniques have been studied in the literature.)

In the Prefetch I algorithm, the home host determines the query keyword *k* that has the shortest inverted list. We assume that hosts have information on keyword frequencies; if not, Prefetch I is not applicable. In phase 1, the home host sends a single subquery containing *k* to the host that handles *k*. When the home host receives the partial answer set, it starts phase 2, which is the same as in the un-optimized algorithm, except that the partial answer set is attached to all subqueries. Before a host returns its partial answer set, it intersects it with the partial answer set of the phase 1 subquery, which reduces the size of the partial answer sets that are returned in phase 2.

6 Experimental Parameters

In this section we summarize two studies we have performed to evaluate the index partition and query processing trade-offs. We believe they are representative of the types of analysis that needs to be performed to evaluate physical design alternatives for information retrieval. In particular, we focus on the experimental parameters used and their impact on response time. Our ranking of these parameters gives an overview on the important areas to consider when designing an information retrieval system. In addition to the simulation work described here, a general interest in the performance of text document retrieval systems has led to a standardization effort for benchmarking of systems [4].

The first study [25] focused on full-text information retrieval. In full-text retrieval, the inverted index contains essentially the same information as the documents, since the position of each word in each document is recorded. Our inverted list model was based on experimental data, and our query model was based on a probabilistic equations. The second study [23] focused on abstracts text information retrieval where each electronic

Parameter	Base Value	Influence
Database scale	1.0	-359.6
Fraction of query words which are striped	0.0	278.4
Disk bandwidth (Mbit/sec)	10.4	112.7
Compression ratio	0.5	-67.4
Multiprogramming level (per host)	4	-48.1
CPU speed (MIPS)	20.0	47.7
Posting size (bits)	40.0	-44.5
Hosts	1	-27.9
Disks per I/O bus	4	25.4
I/O bus bandwidth (Mbit/sec)	24.0	11.2
Buffer overhead (ms)	4.0	-9.33
Disk buffer size (Kbyte)	32	9.12
LAN bandwidth (Mbit/s) (4 hosts)	100.0	2.33
I/O bus overhead (ms)	0.0	-1.96
Disk seek time (ms)	6.0	-1.93
Bytes per block	512	-0.81
Instructions per byte for a merge	40	0.0
Answer entry size (bytes)	4.0	0.0
Instructions per byte of decompression	40	0.0
Instruction count per query	500,000	0.0
Cache size (postings)	0	0.0
Instructions per byte of union operation	5	0.0
Subquery instruction count	100,000	0.0
Instructions per disk fetch	10,000	0.0
LAN overhead (ms)	0.1	0.0
LAN bandwidth (Mb/s)	100.0	0.0
Subquery length (bytes)	1024	0.0

Table 3: A ranking of the influence of simulation parameters on response time for the system index organization with Prefetch I query optimization.

abstract is an abstract of a paper document. In this form of retrieval, the inverted index records only the occurrence of a word in an abstract, and not every occurrence. This dramatically reduces the size of the index with respect to full-text retrieval.

In general, our results indicate that the host index organization is a good choice, especially if long inverted lists are striped across disks. Long inverted lists are present in full-text information retrieval. Since the lists are long, the bottleneck is I/O performance. The host index organization uses system resources effectively and can lead to high query throughputs in many cases. When it does not perform the best, it is close to the best strategy.

For an application where only abstracts are indexed, the system organization (with the Prefetch I optimization) actually outperforms the host organization. The bottleneck for these systems is the network. This is because the inverted lists are much shorter, and can be easily moved across machines.

To study the impact of the experimental parameters on response time, we focus on the second study. Our inverted list model and query model were based on inverted lists of actual abstracts and traces of actual user queries from the Stanford University FOLIO information retrieval system. In both studies, query processing and hardware measurement where accomplished by using a sophisticated simulation containing over 28 parameters. Table 3 lists the parameters and the default values of each parameter. For each parameter in the table, a simulation experiment was run which linearly varied the values of the parameter. The simulation reflects the the architecture shown in Figure 1, as determined by the number of hosts, I/O buses and disks shown in the table. Full details of our experiments and our results are available in the references.

One way to succinctly show the parameters involved in the studies and their influence on performance is to

"rank" them by their (normalized) influence. Here we only look at query response time as the performance metric. In particular, if a and b are the smallest and largest values measured for a parameter and x is the response time for a and y the response time for b, we compute $(y - x)/(a/b)$ as an estimate of the influence the parameter has on response time. Of course, this measure is only a rough indication of influence. The measure depends on the ranges of values over which a parameter is measured. It also assumes that response time is monotonic over the range of values chosen. We have inspected the data to insure that this last condition holds.

Table 3 shows the ranking of 28 parameters for the system index organization, as described in Section 4, with the Prefetch I query optimization, as described in Section 5. In previous work, the system index organization was shown to be the best overall choice for an index organization for abstracts text information retrieval. The positive or negative nature of the ranking is due to the positive or negative influence the parameter has on response time.

Database scale has the strongest influence – this parameter linearly scales the length of an inverted list and scales the lengths of all other objects in the system – such as the size of the answers to queries. With striping, a fraction of the inverted lists (in particular the longest ones) are striped across the disks within a computer system. This is a complementary technique to the list partitioning done by the basic index organization we have discussed, and can be very beneficial. Disk bandwidth is important due to the disk intensive nature of the computation. The compression ratio linearly scales the length of the inverted lists, but does scale any other parameter. The multiprogramming level is the number of simultaneous jobs which are run on each host. The relative CPU speed scales all computations which compute the number of instructions needed to accomplish a task. The posting size is the number of bits needed to represent a posting. Hosts represents the number of processors in the system. When this parameter is increased, a copy of the processor is made. That is, if the parameter doubles, the number of I/O buses and disks in the entire system also doubles. In addition, the workload doubles, since the number of concurrent queries is allocated on a per host basis. Examining the parameters at the end of the table, we see that within the accuracy of the measurement, several parameters have no influence on response time. One surprising fact shows cache size as having no influence. In fact, caches have no influence on response time, but have a tremendous influence on throughput. Essentially, each query almost always has a cache miss. Thus, the response time of the query is dictated by the read from disk of the cache miss and thus the cache has little influence on response time. However, most queries have cache hits also, which dramatically improves throughput.

7 Conclusion

In this article, we have sampled issues in parallel information retrieval. As an introduction to the issues involved, we have discussed the literature in the area to introduce the various areas of research. We then focused on a specific example to illustrate the issued involved in distributed shared-nothing information retrieval, and discuss physical index organization and query optimization techniques. Then, to give the reader a sense of the important variables in the design of a system, we ranked the various parameters in an experimental simulation study in terms of their influence on the response time of query processing.

References

[1] Ijsbrand Jan Aalbersberg and Frans Sijstermans. High-quality and high-performance full-text document retrieval: the parallel infoguide system. In *Proceedings of the First International Conference on Parallel and Distributed Information Systems*, pages 151–158, Miami Beach, Florida, 1991.

[2] Forbes J. Burkowski. Retrieval performance of a distributed text database utilizing a parallel processor document server. In *Proceedings of the Second International Symposium on Databases in Parallel and Distributed Systems*, pages 71–79, Dublin, Ireland, 1990.

[3] Janey K. Cringean, Roger England, Gordon A. Manson, and Peter Willett. Parallel text searching in serial files using a processor farm. In *Proceedings of Special Interest Group on Information Retrieval (SIGIR)*, pages 429–453, 1990.

[4] Samuel DeFazio. Full-text document retrieval benchmark. In Jim Gray, editor, *The Benchmark Handbook for Database and Transaction Processing Systems*, chapter 8. Morgan Kaufmann, second edition, 1993.

[5] Samuel DeFazio and Joe Hull. Toward servicing textual database transactions on symmetric shared memory multiprocessors. *Proceedings of the Int'l Workshop on High Performance Transaction Systems*, Asilomar, 1991.
[6] Perry Alan Emrath. *Page Indexing for Textual Information Retrieval Systems*. PhD thesis, University of Illinois at Urbana-Champaign, October 1983.
[7] Christos Faloutsos. Access methods for text. *ACM Computing Surveys*, 17:50–74, 1985.
[8] J. Fedorowicz. Database performance evaluation in an indexed file environment. *ACM Transactions on Database Systems*, 12(1):85–110, 1987.
[9] William B. Frakes and Ricardo Baeza-Yates. *Information Retrieval: Data Structures and Algorithms*. Prentice-Hall, 1992.
[10] Gaston H. Gonnet, Ricardo A. Baeza-Yates, and Tim Snider. Lexicographical indices for text: Inverted files vs. PAT trees. Technical Report OED-91-01, University of Waterloo Centre for the New Oxford English Dictionary and Text Research, Canada, 1991.
[11] Lee A. Hollaar. Implementations and evaluation of a parallel text searcher for very large text databases. In *Proceedings of the Twenty-Fifth Hawaii International Conference on System Sciences*, pages 300–307. IEEE Computer society Press, 1992.
[12] Byeong-Soo Jeong and Edward Omiecinski. Inverted file partitioning schemes for a shared-everything multiprocessor. Technical Report GIT-CC-92/39, Georgia Institute of Technology, College of Computing, 1992.
[13] Donald E. Knuth. *The Art of Computer Programming*. Addison-Wesley, Reading, Massachusetts, 1973.
[14] Zheng Lin. Cat: An execution model for concurrent full text search. In *Proceedings of the First International Conference on Parallel and Distributed Information Systems*, pages 151–158, Miami Beach, Florida, 1991.
[15] Patrick Martin, Ian A. Macleod, and Brent Nordin. A design of a distributed full text retrieval system. In *Proceedings of Special Interest Group on Information Retrieval (SIGIR)*, pages 131–137, Pisa, Italy, September 1986.
[16] Gerard Salton. *Automatic Text Processing*. Addison-Wesley, New York, 1989.
[17] Bruce Raymond Schatz. Interactive retrieval in information spaces distributed across a wide-area network. Technical Report 90-35, University of Arizona, December 1990.
[18] Kurt Shoens, Anthony Tomasic, and Hector Garcia-Molina. Synthetic workload performance analysis of incremental updates. In *Proceedings of Special Interest Group on Information Retrieval (SIGIR)*, Dublin, Ireland, 1994.
[19] Craig Stanfill. Partitioned posting files: A parallel inverted file structure for information retrieval. In *Proceedings of Special Interest Group on Information Retrieval (SIGIR)*, 1990.
[20] Craig Stanfill and Brewster Kahle. Parallel free-text search on the connection machine system. *Communications of the ACM*, 29:1229–1239, 1986.
[21] Craig Stanfill, Robert Thau, and David Waltz. A parallel indexed algorithm for information retrieval. In *Proceedings of the Twelfth Annual International ACM/SIGIR Conference on Research and Development in Information Retrieval*, pages 88–97, Cambridge, Massachusetts, 1989.
[22] Harold S. Stone. Parallel querying of large databases: A case study. *IEEE Computer*, pages 11–21, October 1987.
[23] Anthony Tomasic and Hector Garcia-Molina. Caching and database scaling in distributed shared-nothing information retrieval systems. In *Proceedings of the Special Interest Group on Management of Data (SIGMOD)*, Washington, D.C., May 1993.
[24] Anthony Tomasic and Hector Garcia-Molina. Performance of inverted indices in shared-nothing distributed text document information retrieval systems. In *Proceedings of the Second International Conference On Parallel and Distributed Information Systems*, San Diego, 1993.
[25] Anthony Tomasic and Hector Garcia-Molina. Query processing and inverted indices in shared-nothing document information retrieval systems. *The VLDB Journal*, 2(3):243–271, July 1993.
[26] Anthony Tomasic, Hector Garcia-Molina, and Kurt Shoens. Incremental updates of inverted lists for text document retrieval. Technical Note STAN-CS-TN-93-1, Stanford University, 1993. Available via FTP db.stanford.edu:/pub/tomasic/stan.cs.tn.93.1.ps.
[27] Anthony Tomasic, Hector Garcia-Molina, and Kurt Shoens. Incremental updates of inverted lists for text document retrieval. In *Proceedings of 1994 ACM SIGMOD International Conference on Management of Data*, Minneapolis, MN, 1994.
[28] Howard R. Turtle and W. Bruce Croft. Uncertainty in information retrieval systems. In Amihai Motro and Philippe Smets, editors, *Proceedings of the Workshop on Uncertainty Management in Information Systems*, pages 111–137, Mallorca, Spain, September 1992.
[29] Peter Weiss. *Size Reduction of Inverted Files Using Data Compression and Data Structure Reorganization*. PhD thesis, George Washington University, 1990.
[30] Justin Zobel, Alistair Moffat, and Ron Sacks-Davis. An efficient indexing technique for full-text database systems. In *Proceedings of 18th International Conference on Very Large Databases*, Vancouver, 1992.

IEEE TRANSACTIONS ON KNOWLEDGE AND DATA ENGINEERING

CALL FOR PAPERS

Research Surveys and Correspondences on Recent Developments

We are interested to publish in the *IEEE Transactions on Knowledge and Data Engineering* research surveys and correspondences on recent developments. These two types of articles are found to have greater influence in the work of the majority of our readers.

Research surveys are articles that present new taxonomies, research issues, and current directions on a specific topic in the knowledge and data engineering areas. Each article should have an extensive bibliography that is useful for experts working in the area and should not be tutorial in nature. Correspondences on recent developments are articles that describe recent results, prototypes, and new developments.

Submissions will be reviewed using the same standard as other regular submissions. Since these articles have greater appeal to our readers, *we will publish these articles in the next available issue once they are accepted*.

Address to send articles:	Benjamin W. Wah, Editor-in-Chief Coordinated Science Laboratory University of Illinois, Urbana-Champaign 1308 West Main Street Urbana, IL 61801, USA Phone: (217) 333-3516 (office), 244-7175 (sec./fax) E-mail: b-wah@uiuc.edu
Submission Deadline:	None
Reviewing Delay:	One month for correspondences, three months for surveys
Publication Delay:	None; articles are published as soon as they are accepted
Submission Guidelines:	See the inside back cover of any issue of *TKDE* or by anonymous ftp from manip.crhc.uiuc.edu (128.174.197.211) in file /pub/tkde/submission.guide.ascii
Length Requirements:	40 double-spaced pages for surveys, 6 double-spaced pages for correspondences
Areas of Interest:	See the editorial in the February'94 issue of *TKDE* or by anonymous ftp from manip.crhc.uiuc.edu in file /pub/tkde/areas.of.interest

IEEE Computer Society
1730 Massachusetts Ave, NW
Washington, D.C. 20036-1903

Non-profit Org.
U.S. Postage
PAID
Silver Spring, MD
Permit 1398