Dynamic File Allocation in Disk Arrays

Gerhard Weikum, Peter Zabback, and Peter Scheuermann*
ETH Zurich
Department of Computer Science
Information Systems – Databases
CH-8092 Zurich, Switzerland
E-mail: {weikum, zabback}@inf.ethz.ch, peters@epsilon.eecs.nwu.edu

Abstract

Large arrays of small disks are being considered as a promising approach to high performance I/O architectures. In this paper we deal with the problem of data placement in such a disk array. The prevalent approach is to decluster large files across a number of disks so as to minimize the access time to a file and balance the I/O load across the disks. The data placement problem entails determining the number of disks and the set of disks across which a file is declustered. Unlike previous work, this paper does not assume that all files are allocated at the same time but rather considers dynamic file creations. This makes the placement problem considerably harder because each placement decision has to take into account the current allocation state and the access frequencies of the disks and the existing files. As a result, file creation may involve partial reorganization on one or more disks. The paper proposes heuristic algorithms for the placement of dynamically created files. The algorithms provide a good compromise between maximizing I/O performance of the disk array and minimizing the work invested in partial reorganizations. The paper presents preliminary performance results of various alternative algorithms under a synthetic workload.

1 Introduction

To meet the requirements of high performance I/O architectures, disk arrays [23] are being considered as a promising approach. In a disk array, a large number of small disks are used rather than relatively few large disks, and a very high bandwidth interconnect is used for the data transfer between disks and memory. Such an architecture not only provides a higher I/O bandwidth, but has also lower costs compared to large disks. The high bandwidth is achieved because a disk array has many arms so that more I/Os can be processed in parallel. This I/O parallelism can be exploited in two different ways:

1. A higher number of independent I/Os can be performed in parallel. While this does not significantly improve the response time of small I/Os, it can improve the I/O throughput drastically.
2. The response time of large I/Os (i.e., I/Os that request many consecutive blocks) can be decreased by orders of magnitude. Suppose, for example, an entire relation is read into memory in order to compute a join. If the requested relation is distributed over multiple disks, then all fragments of the relation can be read in parallel. This sort of distribution is also known as *declustering* or *striping*.

In this paper, we deal with software-controlled disk arrays, also known as independent drive disk arrays [15], disk farms, or simply multi-disk systems [18]. In this type of disk array, multiple disks can cooperate on a single data request while, at the same time, other requests can be directed to a single disk. The file system is responsible for the distribution of the data and the disk load balancing. Thus, the placement of files can be tailored to the application. We believe that file systems and database systems, which often have highly skewed data access distributions, crucially need the flexibility of software-controlled disk arrays for load balancing.

1.1 The Data Placement Problem

Data placement within a disk array is a crucial issue for both applications with mostly small I/Os and applications with large I/Os. For small I/Os, load balancing is the critical issue. For large I/Os, minimizing the access time and load balancing are equally important objectives. In order to exploit the advantages of a disk array, we have to address the following issues.

- *Type of declustering unit*:
  Declustering can be based on relations (i.e., sets of records) or files. In the first case, a relation is partitioned into fragments, e.g., based on key ranges, and the fragments are allocated on different disks. In the second case, a file is partitioned into runs of consecutive blocks, and these runs are allocated on different disks.
  In this paper, we consider only the second case for the following reason. In applications that deal with complex objects such as office documents [38], it may be desirable to decluster a single object that is stored in a large set of blocks. This case can be dealt with by handling the large complex object as a separate file (not necessarily in the sense of a separate OS file but rather as a separate set of blocks that constitute a "storage cluster" [6],[13],[14],[17],[27]). On the other hand, for relations that consist of many small records, the mapping of records into the blocks of the underlying files may be important to reflect a sort order of records. This mapping can be defined in an additional design step on top of file declustering. If a relation is stored physically sorted in the blocks of the underlying file(s), then key-range quer-

*Permanent address: Department of Electrical Engineering and Computer Science, Northwestern University, Evanston, Illinois 60208-3118. This work was performed while the author was visiting ETH Zurich.

ies, for example, translate into byte-range accesses which can be parallelized by file declustering. Using files as the declustering unit promises to deal with different sorts of objects in a uniform way, and it is potentially useful for both database systems and OS file systems.

- *Degree of declustering*:
  Determining the number of disks across which a relation or file is declustered is a critical performance issue. Small files (e.g., smaller than a track) are obviously better off if they are entirely allocated on one disk, since this allows reading or writing the entire file or any portion of it in a single set-oriented I/O [35]. For large files, a high degree of declustering increases the I/O parallelism, but it also increases the maximum seek time and maximum rotational delay among the disks that serve a read or write request. In addition, the CPU costs of processing a request may increase with the degree of declustering as more device driver work has to be performed. Of course, these disadvantages do not apply to small I/Os that read or write only a single block of the file. So the optimal degree of declustering of a file depends on both the size of the file and its access characteristics.
- *Load balancing*:
  When we have decided to decluster a relation or file across w disks, and if w is smaller than the number of disks in the disk array, we still have the problem of selecting the w disks on which the relation or file is actually placed. An intelligent selection is crucially important for disk load balancing. This can be achieved by taking into account statistics (or estimates) of the access frequencies of disks and files. In practice, disk load balancing is achieved by hand tuning the data placement periodically [18]. Automating data placement would be an important step toward self-tuning systems, saving an enormous amount of system administrator work.
- *Dynamic allocation and reorganization:*
  The data placement problem is already hard if one assumes that all relations or files are allocated at the same time. In practice, however, this is an unrealistic assumption. File systems are highly dynamic, and many database system applications require the dynamic creation (or deletion) of relations. Moreover, during the evolution of an application, relations or files may grow quite a bit. The resulting dynamic data placement problem may in turn require *partial reorganizations* on heavily utilized disks. Finally, long-term changes of access frequencies may require *global reorganizations* on the entire disk array.

This paper addresses the *dynamic data placement* problem for files. This entails determining the optimal degree of declustering and selecting the set of disks across which a file is declustered. Our main contributions are the following:

- Given an average request size (number of blocks) for a file, and the throughput requirements (number of requests per second) for the disk array, we present a method for deriving the file's "optimal" degree of declustering such that the response time of requests is minimized. This derivation is based on a simple analytical model for response time and throughput as functions of the degree of parallelism in serving a request (see [9] for similar considerations).
- Given a file's degree of declustering, we propose algorithms for selecting the disks across which the file is declustered such that the overall load is balanced across all disks. Because we deal with dynamic file creations and expansions, we consider various sorts of partial reorganizations. The developed algorithms are based on heuristics, since even the static data placement problem is known to be NP-complete [4]. For space limitations, we do not consider global reorganizations in this paper.

Note that our approach to load balancing is independent of how a file's degree of declustering is determined, and what type of declustering unit is used. Thus, our algorithms are also useful for declustering based on relations, in a shared-nothing architecture where the disk load balancing also affects the CPU load balancing, or in disk arrays with workload assumptions that differ from ours.

1.2 Related Work

Declustering (or "striping") data across multiple disks has originally been proposed in [16] and [25]. The performance studies in [18] and [24] show a significant improvement for different sorts of workloads and different sorts of multi-disk architectures. The impact of the degree of declustering on I/O throughput has been studied in [3].The idea of disk arrays was first introduced in [23]. Disk arrays have motivated the design of novel file systems (e.g., [32]), and will presumably influence the file systems of several computer vendors ([12],[20]). However, none of the proposed file systems seems to be flexible enough to cope well with conventional on-line transaction processing as well as accesses to complex objects. The design of our own file system FIVE has been influenced by some ideas introduced in XPRS [32].

Declustering, for performance as well as availability reasons, is employed in several database systems, including commercial systems such as NonStopSQL [33] and Teradata [34] as well as research prototypes such as Gamma [7] and Bubba [2]. These systems use relation fragments as the declustering unit; the partitioning schemes include round-robin, hash partitioning, key-range partitioning, and a hybrid of the latter two ([8],[9]). Other database systems support similar partitioning schemes but require the database administrator to allocate space on different disks manually.

The data placement problem (i.e., determining across how many disks and across which disks a relation or file is declustered) has been addressed only in the Bubba project [4]. The degree of declustering is chosen based on an analytical model that estimates the performance for a given degree of declustering. The disk selection is based on the notion of *heat*, which is the estimated access frequency of a relation. To achieve good load balancing, disk-resident relations are placed

in descending heat order on the disks with the smallest accumulated heat. Note that this heuristic algorithm assumes that all relations are allocated at the same time so that the allocation itself does not pose any problems.

None of the mentioned work considers the dynamic creation of files or relations and the possibly necessary reorganizations. Previous work on reorganization in both file systems (e.g., [26]) and database systems (e.g., [14],[21],[28]) has concentrated on the reorganization of a single disk for the purpose of compaction or improving the clustering of records and blocks.

The work presented here aims at automating the data placement and reorganization decisions in a disk-array-based file system. It is most closely related to the data placement work in Bubba [4]. However, it extends this work in that we do not assume that all files are allocated at the same time but rather allow dynamic file creations and expansions.

1.3 Outline of the Paper

The rest of this paper is organized as follows. In Section 2, we give a short description of our prototype file system called FIVE, which serves as the experimental platform for our work. In Section 3, we discuss the performance tradeoffs in dynamic data placement. In Section 4, we develop heuristic data placement algorithms for dynamically created files. In Section 5, we present preliminary simulation results on dynamic file creations. Finally, we conclude with an outlook on future work on data allocation and reorganization.

2 The File System: FIVE Concepts

We have developed an experimental file system called FIVE that is supposed to exploit the potential advantages of disk arrays. FIVE stands for File System with Adaptive Enhancements. As one might guess, FIVE is based on five concepts, namely blocks, runs, extents, regions, and files.

The smallest unit of data that is managed by FIVE is a *block*. A block is a contiguous fixed-length area on one disk. It is the minimal transfer unit between disk and memory. We require that the block size is the same for all disks and all files. Note that this requirement is not really restrictive because, by using set-oriented I/O, multiple blocks can be transferred in one I/O operation.

A physically contiguous collection of logically consecutive blocks of a file is a *run*. A run consists of the largest number of logically consecutive blocks that reside on one disk. The run size is the "striping granularity" in the sense of ([3],[12]); a number of logically consecutive runs that reside on different disks form a "stripe".

A physically contiguous collection of one or more runs on one disk is called an *extent*. An extent is described by its disk number, its start address, and its size in blocks. A collection of extents located on different disks is called a *region*. All extents of a region have the same run size and the same number of runs per extent. Thus, a region is described by the run size, the extent size, the number of extents, and a list of the start addresses of the extents (i.e., disk number and address of the first block).

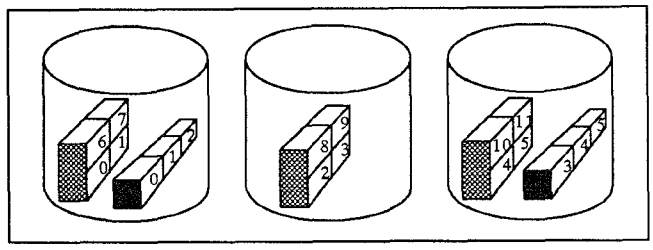

Figure 1: File organization in FIVE

Finally, a *file* is stored as a collection of regions. When a file is created, its space is allocated in one region. When a file grows during its lifetime, the file is expanded by allocating additional regions. If the growth factor of a file can be estimated in advance, larger regions can be allocated in order to avoid excessive expansions.

The net effect of this file organization is that runs are placed on w different disks in a round-robin fashion, with those runs that end up on the same disk being clustered together in one extent. The example in Figure 1 illustrates the five introduced concepts. The figure shows the physical allocation of two files and their logical block numbering. Each of the two files consists of one region. The grey region consists of three extents, each of which consists of two runs that are vertically stacked in the figure. The run size of this region is 2 blocks. The black region of the other file consists of two extents, each of which consists of one run of 3 blocks.

The number of extents of a region is called the *width* of the region. The number of blocks of a run is called the *depth* of the region. The number of runs of an extent is called the *height* of the region. Thus, width times depth times height yields the total number of blocks of the region. In the example of Figure 1, the grey region has width 3, depth 2 and height 2; the black region has width 2, depth 3, and height 1.

Logical block numbers are translated into disk block addresses through the *region table* of a file. We assume that all region tables of frequently used files fit into main memory so that I/Os for region table look-ups are negligible. Note that the separation of logical block numbers and disk block addresses is a prerequisite for all sorts of reorganizations such as relocating extents or combining multiple regions.

In addition to the region tables and a free space bit map for each disk, FIVE keeps the following free space information for each disk d_i:

- *freelist*(d_i): a list containing the addresses of free areas of disk d_i. The list is kept in decreasing size order (i.e., largest free area first).
- *free*(d_i): the total number of free blocks on disk d_i. This is not necessarily contiguous space. *free*(d_i) is the sum of the sizes of all areas in *freelist*(d_i).
- *maxfree*(d_i): the largest contiguous area on disk d_i. *maxfree*(d_i) corresponds to the first entry of *freelist*(d_i).

The workload of the system is reflected in a metric called *heat* [4], where the heat of an object is the access frequency of the object over some period of time (e.g., one day or week). Only accesses that result in disk I/O are counted so as to factor out the effect of caching. The following heat statistics are (continuously or periodically) collected while the system is running.

- $H(e)$: the heat of an extent e, i.e., the sum of the access frequencies of the extent's blocks.
- $H(d_i)$: the heat of a disk d_i, i.e., the accumulated heat of the extents of the disk.

In addition to heat, the *temperature* of an object is another useful metric, where temperature is defined as follows [4]:

- $T(x) = \frac{H(x)}{size(x)}$: the temperature of an extent, or file x.

  This metric reflects both the access frequency and the size of an object. Thus, for a large object to have the same temperature as a small object, the large object must be accessed more frequently.

Our file system FIVE maintains the following two sorted lists that are used in the data placement algorithms:

- *heatlist*: a global list of the disks in increasing heat order ($H(d_1) \leq H(d_2) \leq H(d_3) \leq \ldots \leq H(d_N)$). This list is also used to calculate the average disk heat *avg_heat*.
- *templist*(d_i): for each disk d_i, a list containing the addresses of allocated extents on d_i in decreasing temperature order.

3 Performance Tradeoffs

The following performance goals have guided the development of our dynamic data placement algorithms.

- *Minimal access time to consecutive portions of a file*, i.e., minimizing the response time for read and write requests.
- *Balanced disk load*, i.e., distributing the load across the disks as uniformly as possible. This is an important goal for both I/O throughput and file access response time because it minimizes queueing delays. Note that disk load balancing is a crucial issue for both shared-nothing systems [31], where the disk load balance directly affects the CPU load balancing, and shared-memory systems, where disk load balancing and CPU load balancing are orthogonal issues.
- *Minimal costs of partial reorganizations*, i.e., minimizing the extra disk load caused by reorganization steps that result from certain placement decisions. Such placement decisions are in turn a consequence of the other two performance goals. For example, if an extent is placed on a particular disk for load balancing reasons and this disk does not have enough contiguous space, then a partial reorganization is necessary to reclaim sufficient free space.

Note that we do not aim at tuning the data placement for maximum throughput, i.e., maximizing the throughput at which the disk array saturates, because this would be at the expense of response time. In practice, disks are often underutilized (e.g., with 50 % utilization) on purpose to guarantee acceptable response time (see, e.g., [11],[29]). Throughput requirements are taken into account by limiting the number of disks that serve a single read/write request so that the required request rate can be satisfied (see Section 4.1).

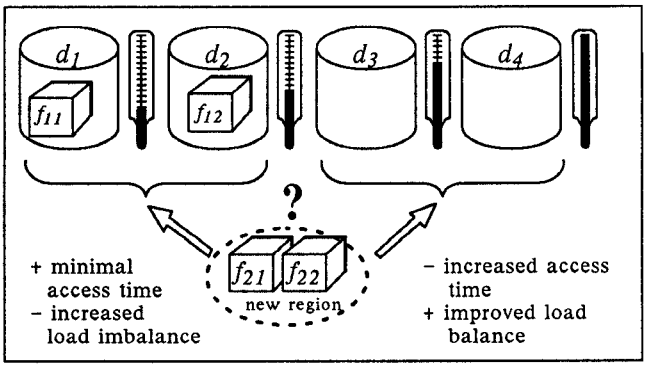

Figure 2: Incompatible performance goals

The above three performance goals may be incompatible in some situations, as illustrated in the example shown in Figure 2. The example shows a file f that consists of one region with two extents f_{11} and f_{12}. Suppose that the file is expanded by allocating a new region with two extents f_{21} and f_{22}, and suppose that reading the entire file is the dominating operation. Typically, the file access time would become minimal if the new extents were placed on disks d_3 and d_4, which do not yet hold any extents of file f. However, this would even increase the load imbalance between the two cooler disks and the two hotter disks. On the other hand, placing the two new extents on disks d_1 and d_2 sacrifices the goal of optimal file access performance. A similar tradeoff exists between the first two performance goals, on the one hand, and the goal of minimal reorganization costs, on the other hand. The algorithms in Sections 4 aim at providing a good compromise with respect to our three performance goals.

4 Algorithms for File Creation

In this section, we present algorithms for the dynamic data placement problem that is posed by the creation of new files. The algorithms decide 1) across how many disks a file will be declustered (degree of declustering), and 2) on which disks the file will be placed (disk selection for load balancing). Subsection 4.1 deals with determining the degree of declustering. Subsections 4.2 and 4.3 deal with the disk selection problem, i.e., load balancing. For the sake of modularity, we first present two elementary building blocks for partial reorganizations in Subsection 4.2. These building blocks are used in the description of the disk-selection algorithms that are presented in Subsection 4.3. The performance tradeoffs of alternative algorithms are briefly discussed in Subsection 4.4.

4.1 A Method for Determining the Degree of Declustering

When a new file is created, a single region of size S is allocated. We assume that S is provided by the client of the file system, based on an estimate of the expected

file growth. The width w, depth d, and height h of the allocated region are determined as follows, aiming at a minimum access time to the (new) file while satisfying the throughput requirements of the entire disk array.

- Let R be the estimated average request size of the file, i.e., the number of consecutive blocks that are read or written in a single request. Let P be the degree of parallelism in serving such a request, i.e., the number of disks across which the requested blocks are declustered. We developed a formula (disregarding queueing effects) for the (single-user) response time of the request as a differentiable function $A(P)$ of P [37]. Solving the equation $\frac{dA}{dP} = 0$ yields the optimum degree of parallelism P_{opt}. With less parallelism, the potential for reducing the transfer time of the request is not fully exploited; with a higher degree of parallelism, the additional gain in transfer time is outweighed by the additional increase of the maximum seek and rotational latency of the involved disks.
- [37] also contains a formula, along the lines of [12], for the disk array's maximum throughput τ as a function of the overall average request size $\overline{R}$ and the average degree of parallelism $\overline{P}$. Given a required throughput τ^0, we can solve the inequation $\tau \geq \tau^0$, yielding the maximum average degree of parallelism $\overline{P}_{max}$ as a function of τ^0 and $\overline{R}$.
- Now, our heuristic approach to minimizing response time while observing the throughput requirements is to view $\overline{P}_{max}$ as an upper limit for P_{opt}. Thus, for a file with request size R, we choose the following effective degree of parallelism:

$$P_{eff} = \min (P_{opt}, \overline{P}_{max}, N)$$

  where N is the number of disks in the disk array.
- To achieve a degree of parallelism P_{eff} for a request size R, we have to decluster all consecutive portions of size R over P_{eff} different disks, so that P_{eff} runs can be accessed in parallel. This is achieved by choosing an appropriate depth d for the region that is to be allocated, according to the formula

$$d = \left\lceil \frac{R}{P_{eff}} \right\rceil .$$

  Of course, for larger requests to the same file, the degree of parallelism should be higher. This is achieved by choosing the degree of declustering of the entire file as large as possible. Also, a degree of declustering that is higher than the degree of parallelism of a single request allows parallelism between multiple independent requests to the same file. This consideration yields the following formula for the region width:

$$w = \min \left(N, \left\lceil \frac{S}{d} \right\rceil \right).$$

  Finally, we obtain the height of the region according to the formula:

$$h = \left\lceil \frac{S}{w \times d} \right\rceil .$$

Note that the above result affects only the degree of declustering of a file. The disk selection algorithms that are described below do not depend on how the values for w, d and h are determined. For example, if we wanted to maximize the saturated disk throughput rather than minimize file access response time, the optimal value of w would probably be smaller, but our disk selection algorithms would still be appropriate for load balancing.

Note that the allocated region of size $w \times d \times h$ may be larger than the requested size S. Such an overallocation is unavoidable if one wants to have all runs and all extents of a region to have the same size. While this property keeps the region table small and makes the addressing of blocks efficient, it has the disadvantage that the internal fragmentation of a file (i.e., *1 - file size / allocated space*) can be relatively high. Throughout this paper, however, we consider disk space as an ample resource.

4.2 Building Blocks for Partial Reorganizations

When a dynamic data placement algorithm decides to allocate an extent of a file or region on a particular disk, a partial reorganization may have to be performed on this disk for one of the following two reasons:

A) There is not enough contiguous space available to satisfy the allocation request.

B) The allocation of the new extent would cause the disk to become too hot (because the extent belongs to a hot file) and may thus adversely affect the load balance in the disk array.

Case A) is addressed by performing a partial disk compaction, and case B) is addressed by performing a reorganization step that we call "disk cooling". These two building blocks are described in the following two subsections.

4.2.1 Partial Disk Compaction

A partial disk compaction is necessary when a disk d_i has enough free space for a new extent e but does not have enough contiguous space (i.e., the condition $max\text{-}free(d_i) < size(e) \leq free(d_i)$ holds). In performing the compaction, the amount of data that is moved should be minimized. Since this is essentially a knapsack problem, we employ a relatively simple heuristic algorithm. In a first step, two addresses *low* and *high* are determined such that the total size of the free areas between *low* and *high* is greater than or equal to $size(e)$ and the total size of the existing extents between *low* and *high* is minimized. In the second step all extents between *low* and *high* are shifted in order to reclaim a sufficient contiguous free area.

4.2.2 Disk Cooling

When a disk becomes too hot so that it causes significant load imbalance in the disk array, the heat of the disk can be reduced by moving some of its data to a cooler disk. It is reasonable to move only entire extents so as not to interfere with the goal of minimizing the file access time. For the same reason, an extent is usually not moved to a disk that already holds an extent of the same region (because this would effectively change the width of the region). If a file consists of mul-

tiple regions, one could even exclude disks that already hold an extent of the same file.

Determining the best migration candidates for disk cooling is essentially a knapsack problem, since we want to minimize the amount of data that is to be moved while ensuring that the disk heat will be reduced by a specified amount. A heuristic criterion for selecting the extent(s) to be moved is the temperature $T(e)$ of an extent e. In contrast to the heat metric, the temperature reflects both the benefit and the cost of the reorganization, where the benefit is the achieved decrease of the disk's heat and the cost is proportional to the size of the moved extent(s). A disk cooling algorithm that implements these heuristics is given in [37]. The algorithm keeps cooling the disk (i.e., moves extents to cooler disks) until the heat of the disk drops below a given threshold. The algorithm terminates prematurely if the movement of an extent would cause a cooler disk to become hotter than the disk that is to be cooled.

4.3 Disk Selection Algorithms

In this subsection, we present alternative algorithms for selecting the disks across which a newly created file is placed. The alternatives reflect different priorities of the performance goals described in Section 3. In determining the set of disks across which the file is declustered, our algorithms are driven by information about the heat of disks (i.e., the heatlist) and the allocation states of disks as described below. The alternatives use both sorts of information, but with different priorities.

```
Input:   region width w
         extent size e = d*h (depth d, height h)

result_set := {}; escalation_flag := false
start_label:
for each class C of allocation states (in preference order)
do
    for each eligible C disk d_i in heatlist do
        if d_i not in result_set or escalation flag is set
        then
            if H(d_i) > avg_heat + tolerance
                and there exists a cooler disk d_t that is
                not yet in result_set
            then perform disk cooling algorithm
            fi
            case (allocation state of d_i)
            CS:   allocate extent
            ES:   execute partial disk compaction
                  (d_i, e) and allocate extent
            NES:  perform space reclamation algorithm
                  on d_i and allocate extent
            esac
            result_set := result_set ∪ {d_i}
            if enough extents allocated then exit fi
        fi
    od
od
if not enough extents allocated then
    set escalation flag (i.e., drop the constraint that the
    allocated extents reside on different disks)
    goto start_label
fi
                                              optional
                                              step
```

Figure 3: Generic disk selection algorithm

A straightforward approach to selecting w disks is to pick the w coolest disks by looking up the first w entries of the heatlist. This approach, which is adopted from the data placement algorithm of Bubba [4], seems to be a good heuristic for load balancing. However, in a disk array with dynamic file creations, the disk selection also has to take into account the current allocation states of the disks, where the *allocation state of a disk d_i with regard to a new allocation request of size e* is one of the following states:

- CS: contiguous space available, i.e., $e \leq maxfree(d_i)$
- ES: enough space available, i.e., $maxfree(d_i) < e \leq free(d_i)$
- NES: not enough space available, i.e., $e > free(d_i)$

Obviously, disks with allocation state CS are preferable over ES disks which in turn are preferable over NES disks. Picking an ES disk for an extent allocation requires a partial disk compaction. Picking an NES disk for an extent allocation requires moving some data to a different disk so as to reclaim sufficient free space. Such a *space reclamation* can be accomplished by an algorithm similar to the disk cooling algorithm. Like the disk cooling algorithm, one or more extents have to be moved to one or more cooler disks that do not yet hold an extent of the region or file to which the moved extent belongs. This rule prevents negative effects on load balancing and file access performance. Unlike the disk cooling algorithm, the criterion for selecting the extents that are moved is size (e.g., best-fit) rather than temperature. Also, the termination condition is based on the amount of space that is to be reclaimed rather than the decrease of the disk heat.

Given the additional information about disk allocation states, we obtain various disk selection algorithms by grouping the three possible allocation states (or a subset of them) into classes of equally preferable states and defining a preference order between the classes. Then, a generic disk selection procedure, which is shown in Figure 3, runs as follows. The heatlist is processed in multiple rounds. For each class of states, in preference order, we make a pass over the heatlist and select all eligible disks that are in this class. The allocation states within the same class do not affect whether a disk is selected or not, but they may require different reorganization steps on the selected disk. Note that each round may change the allocation state of some disks. The disk selection procedure terminates when it has selected the required number of disks. If, due to unusual space allocation conditions, not enough disks could be selected after all rounds, then we drop the constraint that the extents of a region must reside on different disks and retry the whole procedure.

The above considerations lead to a family of alternative algorithms, one for each possible grouping of allocation states and preference ordering between the resulting classes. The reasonable alternatives are shown in Figure 4. Of the seven possible alternatives, the following three are considered as viable options:

- Alternative 1 – *Heat Balancing (HB)*: The first w entries of the heatlist are selected, regardless of their allocation states.
- Alternative 5 – *Space-restricted Heat Balancing (SHB)*: NES disks are selected only if there are not enough CS or ES disks. This may be reasonable since the space reclamation on an NES disk involves additional disks, whereas a partial disk compaction on an ES disk is a purely local operation.
- Alternative 7 – *Cost Minimization (CM)*: The heatlist is processed in three rounds, aiming at minimal reorganization costs.

1:	{CS, ES, NES}	All disks are selected according to the heatlist.
2:	{CS, ES}	NES disks are not selected.
3:	{CS}	ES or NES disks are not selected.
4:	{CS} < {ES}	CS disks are selected in the first round ES disks may be selected in the second round; NES disks are not selected.
5:	{CS, ES} < {NES}	CS or ES disks are selected in the first round; NES disks are selected in the second round.
6:	{CS} < {ES, NES}	CS disks are selected in the first round; ES or NES disks are selected in the second round.
7:	{CS} < {ES} < {NES}	CS disks are selected in the first round; ES disks are selected in the second round; NES disks are selected in the third round.

Figure 4: Possible allocation state classes and preference orders

In some alternatives, a relatively hot CS disk may be selected whereas a cool ES or NES disk will eventually be not selected. Moreover, allocating the new extent on the hot disk will possibly make it even hotter. In order to avoid large load imbalances that may result from this behavior, we propose the following disk cooling technique. If the heat of the selected disk exceeds the average disk heat in the disk array and if there exists at least one cooler disk that has not yet been selected for the allocation of the new file, then we invoke the disk cooling algorithm to reduce the heat of the selected disk. That is, one or more extents are removed from the disk and are reallocated on one or more cooler disks (see Section 4.2.2). Of course, this cooling step is reasonable not only for CS disks, but should be applied also to ES disks if they satisfy the above stated conditions. After this step, the new extent is allocated on the selected disk. Note that it is important to perform the disk cooling algorithm before allocating the new extent, because this can save a partial disk compaction if the cooling step turns an ES disk into a CS disk.

4.4 Discussion of Alternatives

In this subsection, we discuss some performance implications of the presented file creation algorithms, using the following example for illustration. Suppose we want to allocate a new file of size S=16 blocks on a disk array containing 12 disks with track size 10 blocks. The average request size to this file is R=10 blocks. Further suppose that our analytical model for the optimal degree of parallelism for such a request yields P_{opt} = 5 and $\overline{P}_{max}$ = 10 (assuming disk parameters from [19], an overall average request size $\overline{R}$=10, and a required throughput of $\tau^0 \approx 50$ requests per second). From this the method described in Section 4.1 derives depth $d=\lceil 10/5 \rceil=2$, width $w=\lceil 16/2 \rceil=8$ as the file's degree of declustering, and height $h=\lceil 16/16 \rceil=1$. That is, a region is allocated consisting of 8 extents each of which consists of one run of 2 contiguous blocks. Now suppose that the heatlist and the current allocation states with regard to a request of size 2 are as shown in Figure 5. Assume that the *avg_heat* of the disks is between $H(d_6)$ and $H(d_7)$. Our three main alternatives for the disk selection would produce the following results:

- *HB*: selects the disks $\{d_1, d_2, d_3, d_4, d_5, d_6, d_7, d_8\}$, which causes partial disk compactions on d_1, d_2, d_4, d_5, and d_7, and space reclamations on d_3 and d_8, and, optionally, disk cooling on d_8.
- *SHB*: selects disks $\{d_1, d_2, d_4, d_5, d_6, d_7, d_9, d_{10}\}$, which causes partial disk compactions on d_1, d_2, d_4, d_5, and d_7, and, optionally, disk cooling on d_9, d_{10}.
- *CM*: selects disks $\{d_6, d_9, d_{10}, d_{11}, d_{12}, d_1, d_2, d_4\}$ (in this order), which causes partial disk compactions on d_1, d_2, and d_4, and may invoke the disk cooling algorithm on d_9, d_{10}, d_{11}, d_{12}.

This example clearly shows the main difference in the performance of the three alternatives. Alternative HB stresses the load balancing goal and is therefore likely to perform more reorganization work. Alternatives SHB and CM, on the other hand, try to avoid partial reorganizations as long as possible and rather sacrifice the load balancing goal.

5 Preliminary Performance Measurements

5.1 Description of the Experiment

We have started investigating the performance of the proposed algorithms. Our testbed consists of the implemented FIVE file system, a load generator, and a simulated I/O system that is based on the C-based, process-oriented simulation language CSIM [30]. The

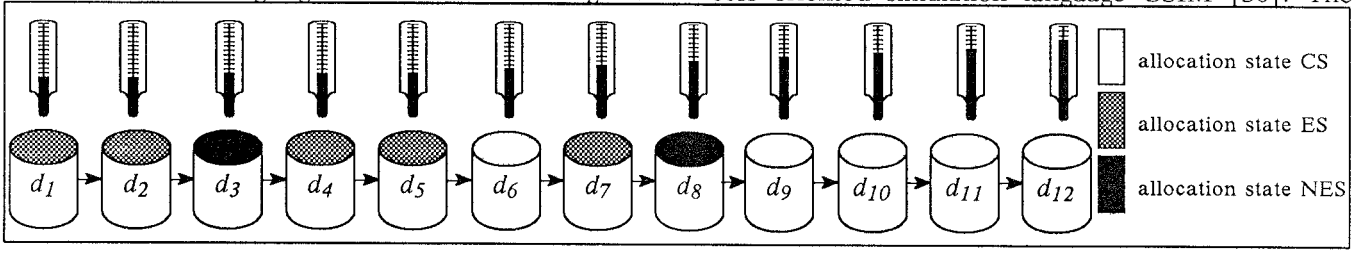

Figure 5: An example of the *heatlist*

simulated disk array consists of 12 (unsynchronized) disks with the parameters shown in Figure 6.

Block Size	4096 Bytes
Track Size	10 Blocks
Tracks per Cylinder	4
Cylinders per Disk	624
Average Rotational Latency	8.3 ms
Average Seek Time	16 ms
Transfer Rate	2.5 MB/s
Cylinder Switch Time	5.0 ms
Head Switch Time	2.5 ms

Figure 6: Parameters of the simulated disks

In a first experiment, which is described here, we concentrated on dynamic file creations in a synthetic complex-object scenario. The test database consists of approximately 2500 complex objects each of which corresponds to one file. The file size is exponentially distributed with an average of 400 KBytes (i.e., 100 blocks of 4KB each). The simulated workload consists of file creations (5% of all operations), file deletions (2.5%), reads (60%), and writes (32.5%). We assume that a complex object is always accessed entirely; that is, the read/write request size is the file size[1]. Files are selected for access according to a 90-10 Zipf-like distribution. That is, 90% of all read/write requests access only 10% of all files. We approximated this skewed access distribution by applying a linear transformation to a normal distribution. The average request size is $\overline{R}$ = 150 blocks. Note that there is a correlation between the heat and the size of a file, as larger files cause more data transfer work.Thus, since the load generation is driven by the skewed heat distribution of files, the average request size is larger than the average file size.

The degree of declustering of a file is determined according to the method of Section 4.1. We assume that the required throughput is τ^0 = 32 requests per second. From this we derive a maximum degree of declustering of 3, using the formula for $\overline{P}_{max}$ that is given in [37]. We have compared the following strategies for dynamic file allocations:

- *Vanilla (V)*: this allocates file extents so as to balance the space utilization of the disks. The V strategy serves as an example of a strategy that disregards load balancing.
- *Cool Vanilla (C-V)*: this may additionally invoke the disk cooling algorithm of Section 4.2. At each create operation, the heat imbalance of the disks is checked; and if the heat of a disk is higher than 1.1 of the average disk heat in the disk array, then the disk cooling algorithm is invoked.
- *Simple Heat Balancing (HB)*: as described in Section 4.3.
- *Cool Heat Balancing (C-HB)*: like HB with the possibility of disk cooling invocations.
- *Simple Cost Minimization (CM)*: as described in Section 4.3.
- *Cool Cost Minimization (C-CM)*: like CM with the possibility of disk cooling invocations.

We generated test databases by running the operation mix of our synthetic workload, starting with an empty database and terminating at a total space utilization of 75%. Such a database was generated for each of the above six strategies, thus constructing six different databases. In addition, we constructed a seventh database from one of the generated ones by statically reallocating all files in descending heat order, that is, by applying the static data placement algorithm of [4]. The seven test databases are referred to as V-DB, C-V-DB, HB-DB, C-HB-DB, CM-DB, C-CM-DB, and OPT-DB. In the actual measurements, our synthetic workload was run against each of these databases at varying arrival rates of requests. Each run consists of 10000 file-system operations.

5.2 Performance Results

Figure 7 shows the standard deviation of the disk heat distribution of the generated test databases. The underlying heat metric is the sum of the number of accesses to the blocks of a disk. A low standard deviation across the disks in the disk array is an indicator for good load balance. The real touchstone for load balancing, however, is the standard deviation of the disk utilizations during the actual measurements. These additional figures, at an arrival rate of 30 requests per second, are also shown in Figure 7.

According to the utilization figures of Figure 7, one would expect that the HB and CM strategies clearly outperform the vanilla strategy, with CM being slightly better than HB. We performed measurements by applying each strategy to the test database that was created by the strategy, referred to as x-DB, and by applying all strategies to the V-DB and the OPT-DB. The measured response time of read and write operations are shown in Figure 8. The main observations from this experiment are summarized in the following.

- Disk cooling was never invoked in the strategies C-HB and C-CM, since the load balancing of HB and CM is already so good that the imbalance threshold for disk cooling invocations was never exceeded during the measurement phase. Thus, C-HB and C-CM perform exactly like HB and CM, respectively. Therefore, the curves for C-HB and C-CM are omitted in Figure 8. For the vanilla strategy, disk cooling had a beneficial effect; that is, strategy C-V clearly outperformed the simple strategy V. For example, at an arrival rate of 30 requests per second, the response time of strategy V (on the V-DB) was 1.66 times higher than that of strategy C-V (on the C-V-DB). On the OPT-DB, V and C-V performed identically; so the C-V curve is omitted in Figure 8 for this case.

1. Even though this is a debatable assumption, we have obtained interesting insights into the behavior of our algorithms. In addition, we are not aware of any published analysis of real-life complex-object accesses, which could serve as a counterpart of, for example, the Berkeley trace data analysis of Unix file accesses [22].

after the database generation phase							
	V-DB	C-V-DB	HB-DB	C-HB-DB	CM-DB	C-CM-DB	OPT-DB
standard deviation	55372	64947	4143	14123	3790	15698	1
during the measurement phase							
	V on V-DB	C-V on C-V-DB	HB on HB-DB	C-HB on C-HB-DB	CM on CM-DB	C-CM on C-CM-DB	CM on OPT-DB
min. utilization	0.80	0.87	0.81	0.81	0.88	0.88	0.88
max. utilization	0.99	0.99	0.97	0.97	0.96	0.96	0.93
avg. utilization	0.91	0.93	0.89	0.89	0.92	0.92	0.90
standard deviation	0.06	0.04	0.04	0.04	0.03	0.03	0.01

Figure 7: Heat distribution and load balance for different strategies

- The vanilla strategies V and C-V both perform well at low arrival rates, as the load imbalance does not yet cause queueing effects. With increasing arrival rate, requests may become queued at a disk. Then, load imbalance causes a dramatic increase of response time because the hottest disk becomes a bottleneck. For example, at an arrival rate of 31 requests per second, the response time of V is almost twice as high as that of the best strategy CM. This observation clearly demonstrates the need for a load balancing strategy.
- At low arrival rates, the HB and CM strategies perform partial disk compactions, without achieving any benefits since request queueing is not yet a problem. This explains why these strategies perform worse than the V strategy which benefits from its space balancing in this case. At higher arrival rates, the reorganizations pay off, for they were actually necessary to improve load balancing and hence decrease the average queue length of the disks.
- Partial disk compaction turned out to be pretty expensive, as it moved up to 4400 blocks. The HB algorithm was far too aggressive, which resulted in excessive invocations of partial disk compactions and high response time, especially on the badly balanced V-DB.

  On the HB-DB and OPT-DB, the problem is that the HB strategy tends to be overly sensitive in that it prefers an ES disk over a slightly hotter CS disk. This situation arises fairly frequently because the disk load is already relatively well balanced so that a few additional requests may already change the order of disks in the heatlist. Obviously, the HB algorithm needs to be augmented by incorporating a heat difference threshold such that approximately equally hot disks are considered as equally preferable.
- The strategy CM performs best at high arrival rates, as it achieves reasonably good load balance and avoids reorganizations as long as possible.

6 Conclusion

In this paper, we have developed algorithms for the dynamic data placement problem in disk arrays. These algorithms can be used for allocating newly created files and for file expansions [37] with the goals of optimal file access time and disk load balancing at acceptably low reorganization costs. Our algorithms are useful for dealing with both advanced DBMS applications such as office document management where a large complex object corresponds to a file, and conventional relations of small records where file declustering can be exploited by an appropriate mapping of records into blocks (i.e., to reflect a sort order). The simulation results for complex-object accesses are a first step toward gaining quantitative insight into the performance of our algorithms. We are planning on a series of performance experiments under different sorts of workloads.

The partial reorganizations of our algorithms are intended to maintain good file access performance and good disk load balancing over fairly long periods of time. In the long term, the performance may nevertheless deteriorate, for example, because of changing file

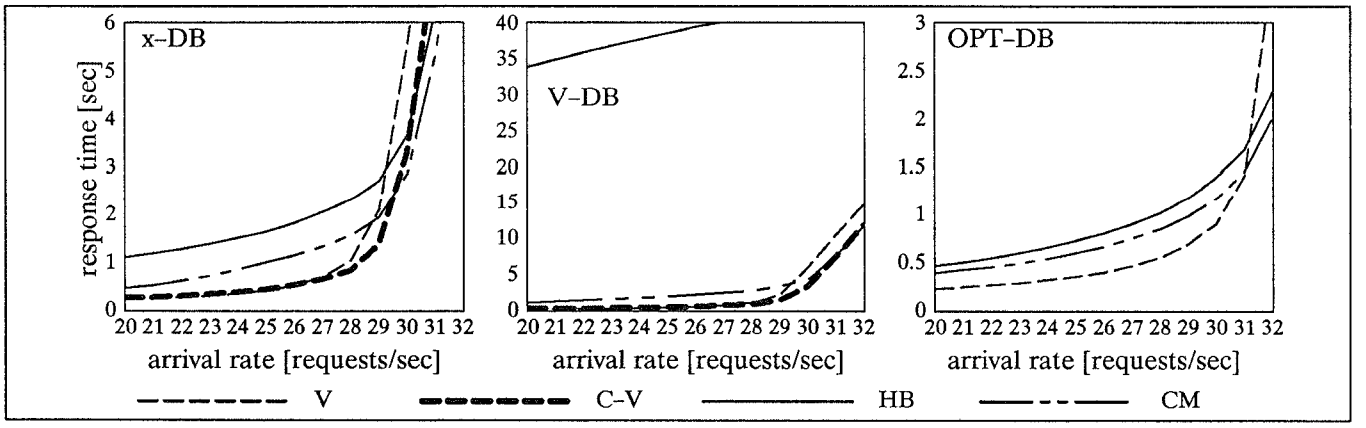

Figure 8: Response time of file creations and read/write requests

access frequencies. Therefore, a global reorganization of the entire disk array will probably be necessary once in a while. We are investigating, when and how such a global reorganization should be performed.

An aspect that we have disregarded so far is reliability. With the increasing number of disks in a disk array, the mean time to failure (MTTF) decreases dramatically. Several solutions to this problem have been proposed, either based on error-correcting codes such as parity blocks [10],[23], or based on replicated data [1],[5]. We plan to investigate to what extent our dynamic data placement algorithms need to be extended if the disk array contains replicated data. We expect that hardware-based schemes such as parity striping on a per sector basis fit well with our approach in that they do neither introduce major complications to the disk selection algorithms nor any degradation of load balancing.

We believe that disk arrays will play an important role in future database system architectures. On the other hand, it is likely that data placement in large disk arrays will not be (easily) manageable for most system administrators. Our long-term goal is a high performance database system that automatically adapts itself to the workload and does therefore not require a human system administrator for performance tuning [36]. Developing algorithms that automate data allocation and reorganization in disk arrays is a crucially important (sub-)problem. We believe that this paper is a promising approach toward solving this problem.

References

[1] Bitton, D. and Gray, J., Disk Shadowing, *14th VLDB Conf.*, 1988

[2] Boral, H., et al, Prototyping Bubba, A Highly Parallel Database System, *IEEE Trans. on Knowledge and Data Engineering*, Vol. 2, No. 1, 1990

[3] Chen, P.M. and Patterson, D. A., Maximizing Performance in a Striped Disk Array, *Proceedings of the 17th Int. Symposium on Computer Architecture*, 1990

[4] Copeland, G., et al, Data Placement in Bubba, *ACM SIGMOD Conf.*, 1988

[5] Copeland, G. and Keller, T., A Comparison of High-Availability Media Recovery Techniques, *ACM SIGMOD Conf.*, 1989

[6] Carey, M.J., et al, Storage Management for Objects in EXODUS, in: Kim, W., Lochovsky, F.H., (ed.) Object-Oriented Concepts, Databases, and Applications, *Addison-Wesley*, 1989

[7] DeWitt, D.J., et al, The Gamma Database Machine Project, *IEEE Trans. on Knowledge and Data Engineering*, Vol. 2, No. 1, 1990

[8] Ghandeharizadeh, S. and DeWitt, D. J., A Multiuser Performance Analysis of Alternative Declustering Strategies, *Proc. of the 6nd Int. Conf. on Data Engineering*, 1990

[9] Ghandeharizadeh, S. and DeWitt, D. J., Hybrid-Range Partitioning Strategy: A New Declustering Strategy for Multiprocessor Database Machines, *VLDB Conf.*, 1990

[10] Gibson, G.A., et al, Failure Correction Techniques for Large Disk Arrays, *Proceedings of the 3rd Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, 1989

[11] Gifford, D. and Spector, A., The TWA Reservation System, *CACM*, Vol.27 No.7, 1984

[12] J. Gray, B. Horst and M. Walker, Parity Striping of Disc Arrays: Low-Cost Reliable Storage with Acceptable Throughput, *VLDB Conf.*, 1990

[13] Hornick, M.F., Zdonik, S.B., A Shared, Segmented Memory System for an Object-Oriented Database, *ACM Trans. on Information Systems*, Vol. 5, No. 1, 1987

[14] Hudson, S. E. and King, R., Cactis: A Self-Adaptive, Concurrent Implementation of an Object-Oriented Database Management System, *ACM TODS*, Vol. 14, No. 3, 1989

[15] Katz, R.H., et al, A Project on High Performance I/O Subsystems, *Database Engineering*, Vol. 11, No. 1, 1988, pp. 40-47

[16] Kim, M.Y., Synchronized Disk Interleaving, *IEEE Trans. on Computers*, Vol. C-35, No. 11, 1986

[17] Lehman, T.J., Lindsay, B.G., The Starburst Long Field Manager, *VLDB Conf.*, 1989

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

[19] M2344K Micro-Disk Drives CE Manual, *Document No. 41FH6817E-01A*, Fujitsu Ltd., 1987

[20] Moad, J., Relief for Slow Storage Systems, *Datamation*, Vol. 36, No. 17, 1990

[21] E. Omiecinski and P. Scheuermann, A Parallel Algorithm for Record Clustering, *ACM TODS*, Vol. 15, No. 4, 1990

[22] Ousterhout, J.K., et al, A Trace-Driven Analysis of the UNIX 4.2 BSD File System, *Proc. ACM Symposium on Operating System Principles*, 1985

[23] Patterson, D.A., Gibson, G., and Katz, R.H., A Case for Redundant Arrays of Inexpensive Disks (RAID), *ACM SIGMOD Conf.*, 1988

[24] Reddy, A.L. and Banerjee, P., An Evaluation of Multiple-Disk I/O Systems, *IEEE Trans. on Computers*, Vol. 38, No. 12, 1989

[25] Salem, K. and Garcia-Molina, H., Disk Striping, *Proc. of the 2nd Int. Conf. on Data Engineering*, 1986

[26] Samadi, B., TUNEX: A Knowledge-Based System for Performance Tuning of the UNIX Operating System, *IEEE Trans. on Software Engineering*, Vol. 15, No. 7, 1989

[27] Schek, H.-J., et al, The DASDBS Project: Objectives, Experiences, and Future Prospects, *IEEE Trans. on Knowledge and Data Engineering*, Vol.2 No.1, 1990

[28] Scheuermann, P., Park, Y., and Omiecinski, E., Heuristic Reorganization of Clustered Files, *Proc. of the Int. Conf. on Foundations of Data Organization*, 1989

[29] Smith, A.J., Input/Output Optimization and Disk Architectures: A Survey, *Performance and Evaluation*, Vol. 1, 1981

[30] Schwetman, H., CSIM Reference Manual (Revision 13), MCC Technical Report ACA-ST-252-87, Rev. 13, *MCC*, Austin, 1989

[31] Stonebraker, M., The Case for Shared Nothing, *IEEE Database Engineering*, Vol. 9, No. 1, 1986

[32] Stonebraker, M., et al, The Design of XPRS, *VLDB Conf.*, 1988

[33] The Tandem Database Group, NonStopSQL: A Distributed, High-Performance, High-Availability Implementation of SQL, *2nd Int. Workshop on High Performance Transaction systems*, Springer, 1989

[34] Teradata, DBC/1012 Database Computer System Manual Release 2.0, *Document No. C10-0001-02*, Teradata Corp., 1985

[35] Weikum, G., Set-Oriented Disk Access to Large Complex Objects, *Proc. of the 5th Int. Conf. on Data Engineering*, 1989

[36] Weikum, G., Hasse, C., Moenkeberg, A., and Zabback, P., The COMFORT Project: A Comfortable Way to Better Performance, *Technical Report, ETH Zurich*, 1990

[37] Weikum, G., Zabback, P., Scheuermann, P., Dynamic File Allocation in Disk Arrays, *Technical Report, ETH Zurich*, 1990

[38] Zabback, P., Paul, H.-B., and Deppisch, U., Office Documents on a Database Kernel - Filing, Retrieval, and Archiving, *Int. Conf. on Office Information Systems*, 1990