High Performance Parallel DBMS

Shahram Ghandeharizadeh, Shan Gao
Department of Computer Science
University of Southern California
Los Angeles, CA 90089, USA
shahram,sgao@cs.usc.edu

Chris Gahagan, and Russ Krauss
BMC Software Inc.
2101 CityWest Blvd.
Houston, TX 77042, USA
chris_gahagan@bmc.com

Abstract

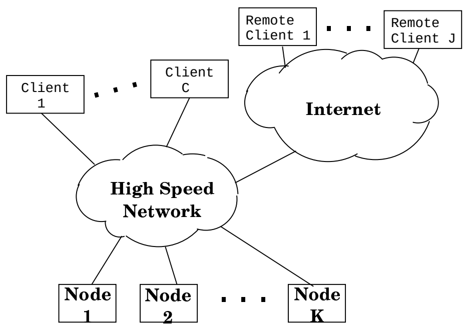
Parallelism is the key to realizing high performance, scalable, fault tolerant database management systems. With the predicted future database sizes and complexity of queries, the scalability of these systems to hundreds and thousands of processors is essential for satisfying the projected demand. This chapter describes three key components of a high performance parallel database management system. First, data partitioning strategies that distribute the workload of a table across the available nodes while minimizing the overhead of parallelism. Second, algorithms for parallel processing of a join operator. Third, ORE as a framework that controls the placement of data to respond to changing workloads and evolving hardware platforms.

1 Introduction

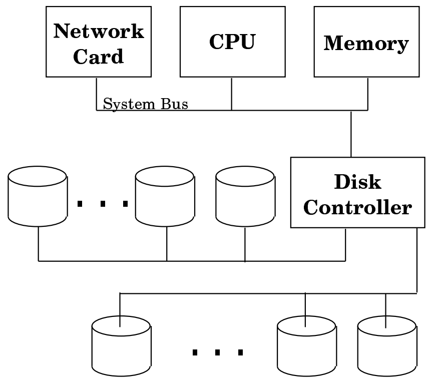
Database management systems (DBMS) have become an essential component of many application domains, e.g., airline reservation, stock market trading, etc. In the arena of high performance DBMS, parallel database systems have gained increased popularity. Example research prototypes include Gamma [23], Bubba [12], XPRS [88], Volcano [54], Omega [34], etc. Products from the industry include Tandem's Non-Stop SQL [89], NCR's DBC/1012 [90], Oracle Parallel Server [74], IBM's DB2 parallel edition [31], etc. The hardware platform of these machines is typically a multi-node platform, see Figure 1.a, where each node might be a computer with one or more disks, see Figure 1.b. In these systems, several forms of parallelism can be utilized to improve the performance of the system. First, parallelism can be applied by executing several queries or transactions simultaneously. This form of parallelism is termed *inter-query* parallelism. Second, *inter-operator* parallelism can be employed to execute several operators in the same query concurrently. For example, multiple processors could execute two or more relational join operators of a complex bushy join query in parallel. Finally, *intra-operator* parallelism can be applied to each operator within a query. For example, multiple processors can be employed to execute a single relational selection operator. This paper describes how a system employs these alternative forms of parallelism.

The placement of data is important for the alternative forms of parallelism This constitutes one focus of this book chapter. In Section 2, we describe the tradeoffs associated with exploiting intra-operator parallelism to execute the selection operator. This has a significant impact on the performance of a complex query. For example, if the appropriate degree of intra-operator parallelism cannot be provided for the selection operator, the performance and degree of intra-operator parallelism of the other operators in a complex query plan may severely be limited. This is specially true for parallel systems that utilize the concept of pipelining and data flow to execute complex queries, i.e., systems such as Gamma [23], Bubba [12], Volcano [54], etc.

Section 3 provides an overview of three alternative algorithms for parallel processing of the join operator: sort-merge, Grace and Hybrid hash-join. We refer the interested reader to [52] for a comprehensive



1.a A multi-node configuration



1.b Sample configuration of one node

Figure 1: Hardware platform of a parallel DBMS.

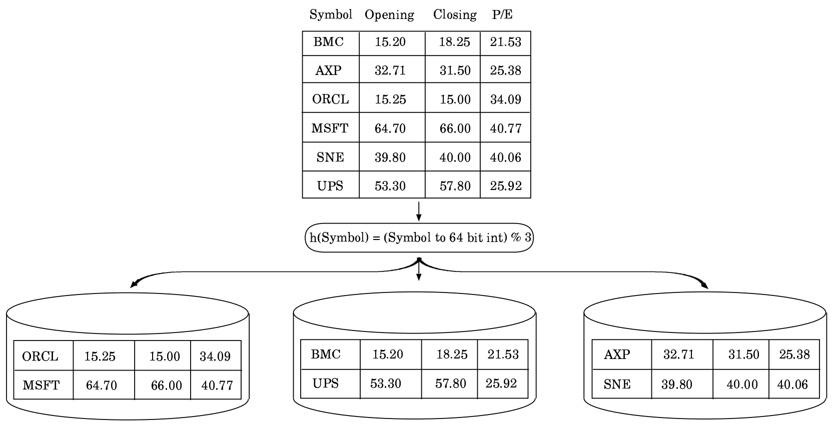
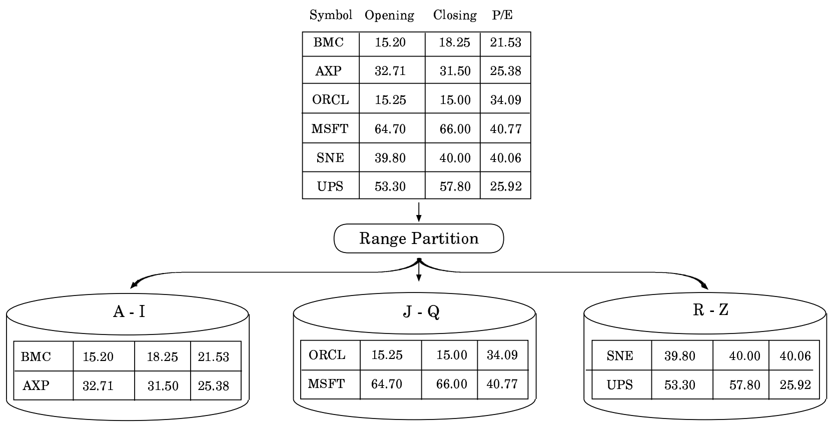


Figure 2: Hash declustering of Stock relation using Symbol attribute.

description of these strategies and other alternatives to process this operator. In Section 4, we describe ORE as a 3 step framework that controls the migration of fragments across the nodes of a parallel DBMS. This section reports on an evaluation of this techniques for both a homogeneous and heterogeneous platform. Brief conclusions and future research directions are offered in Section 5.

2 Partitioning strategies

Multiprocessor database machines utilize the concept of horizontal partitioning [81, 70] to distribute the tuples of each relation across multiple disk drives. The strategy used for partitioning a relation is independent of the storage structure used at each site. The database administrator (DBA) for such a system must consider a variety of alternative organizations for each relation. Three popular partitioning strategies include: 1) round-robin, 2) hash partitioning, and 3) range partitioning. The first strategy distributes the tuples of a relation in a round-robin fashion among the nodes. In effect, this results in a completely random placement of tuples with a uniform distribution of tuples across the nodes. In the hash partitioning strategy, a randomizing function is applied to the partitioning attribute of each tuple to select a home site for that tuple. In the last strategy, the DBA specifies a range of key values for each partition or site. This strategy gives a greater degree of control over the distribution of tuples across the sites.



Symbol	Opening	Closing	P/E
BMC	15.20	18.25	21.53
AXP	32.71	31.50	25.38
ORCL	15.25	15.00	34.09
MSFT	64.70	66.00	40.77
SNE	39.80	40.00	40.06
UPS	53.30	57.80	25.92

BMC	15.20	18.25	21.53
AXP	32.71	31.50	25.38

ORCL	15.25	15.00	34.09
MSFT	64.70	66.00	40.77

SNE	39.80	40.00	40.06
UPS	53.30	57.80	25.92

Figure 3: Range declustering of Stock relation using Symbol attribute.

Figures 2 and 3 show how hash and range partitioning disperse the records of a Stock table across a 3 node configuration. Figure 2 shows a hash function that consumes each record and maps it to one of the disks by: 1) converting its Symbol attribute value from a string into a 64 bit integer, and 2) computing the remainder of this number when divided by 3, the number nodes. This reminder is the disk id and ranges between integer values 0, 1 and 2. Figure 3 shows range partitioning where those records with a Symbol attribute value starting with letter ‘A’ to ‘I’ are assigned to disk 0, ‘J’ to ‘Q’ are assigned to disk 2, and ‘R’ to ‘Z’ are assigned to disk 3. Each piece of the table is termed a fragment. In Figures 2 and 3, each fragment consists of two records. The “Symbol” attribute is termed the partitioning attribute. With range and hash partitioning strategies, the system may direct those queries that reference the partitioning attribute to a single node. For example, the system can direct a query that retrieves the closing price of “BMC” to node 1 with range partitioning. (With hash, this query would be directed to node 2.) This frees up the other two nodes to process other queries.

When a transaction updates the partitioning attribute value of a record, the system might migrate the record from one node to another in order to preserve the integrity of the partitioning strategy. In the example of Figure 3, if the partitioning attribute (Symbol) value of a record changes from “AXP” to “XAP”, the system migrates this record from node 1 to node 3 to preserve the integrity of range partitioning strategy.

In [35], we quantified the performance tradeoff associated with range, hash and round-robin partitioning strategies using the alternative indexing mechanisms provided by the Gamma database machine. This study reveals that for a shared-nothing multiprocessor database machine, no partitioning strategy is superior under all circumstances. Rather, each partitioning strategy outperforms the others for certain query types. The major reason for this is that there exists a tradeoff between exploiting intra-query parallelism by distributing the work performed by a query across multiple processors and the overhead associated with controlling the execution of a multisite query. Localizing the execution of queries requiring minimal amount of resources, results in the best system response time and throughput since the overhead associated with controlling the execution of the query is either minimized or eliminated. On the other hand, for queries requiring more resources, certain tradeoffs are involved. In general, with access methods that result in the retrieval of only the relevant tuples from the disk, if the selectivity factor of the query is very low, it is advantageous to localize the execution of the query to a single processor. While the hash partitioning strategy localizes the execution of the exact match selection queries that reference the partitioning attribute, the range partitioning strategy attempts to localize the execution of all query types that reference the partitioning attribute regardless of their selectivity factor. At the other end of the spectrum, the round-robin partitioning strategy directs a query to all the processors containing the fragments of the referenced relation.

For sequential scan queries, the best response time and throughput is observed when the partitioning strategy constructs the smallest fragment size on each node, the execution of each query is localized to a single node, and the simultaneously executing queries are evenly dispersed across the nodes. The system generally performs best when the query executes all by itself at a site and performs a series of sequential disk requests. By localizing the execution of the query to a single processor, there is a higher probability of maintaining the sequential nature of disk requests made by a query, free from interference of the other concurrently executing queries. Thus, for the sequential scan queries, the optimal partitioning strategy is the range partitioning strategy.

2.1 Multi-attribute partitioning

One may find many alternative multi-attribute partitioning strategies in the literature. These can be categorized into 3 groups based on their objectives. The first strives to optimize the processing of the join operator. Examples include strategies described in [77, 61]. (Section 3.1 details these two techniques.) The second distributes data in a manner so that a selection query performs approximately the same amount of work on each node. Example strategies include Disk Modulo (DM) [28], Fieldwise XOR (FX) [63], Error Correcting Codes (ECC) [33], Coordinate Modulo Distribution (CMD) [69], Hilbert Curve Allocation (HCAM) [32], vector-based declustering [15], Golden Ratio Sequences (GRS) [10]. For a comparison of some of these strategies see [72].

The third group strives to localize the execution of a query that references a partitioning attribute to as few nodes as possible. Example strategies include MAGIC [36, 51] and Bubba's extended range declustering strategy [12]. A comparison of these two techniques is detailed in [46], demonstrating the superiority of MAGIC.

We describe the first group of multi-attribute declustering techniques in Section 3.1 with a description of the hash-join algorithm. While at first glance the remaining two groups might appear contradictory, they complement one another because they are appropriate for different query classes. While the 2^{nd} approach is appropriate for those queries that perform sufficient work at each node to eclipse the overhead of parallelism, the 3^{rd} is appropriate for those that suffer from the overhead of coordinating multi-site queries. Ideally, the later query class should be directed to one node in order to minimize the overhead of parallelism.

In the following, we provide a brief description of the MAGIC declustering strategy. MAGIC differs from the range and hash partitioning strategies in two ways. First, relations are declustered into fragments using several attributes instead of one. Thus, it can restrict the subset of nodes used to execute selection operations on any of the partitioning attributes. Second, the number of fragments and their assignment to processors is determined from the characteristics of the selection operations accessing the relation.

In order to further motivate the MAGIC partitioning strategy, recall the Stock table of Figure 4. Assume one half of the accesses (termed query type A) to the Stock relation use an equality predicate on the Symbol attribute (e.g., select Stock.all from Stock where Stock.Symbol = "BMC") and the remaining queries (termed type B) use a range predicate on the P/E attribute (e.g., select Symbol from Stock where P/E $>$ 10.03 and P/E $<$ 10.09). Furthermore, assume that both queries retrieve only a few tuples. For this workload, the appropriate access methods are a hash index on the Symbol attribute and a B^+ index on the P/E attribute of the STOCK relation. However, either attribute could be selected as the partitioning attribute because both queries have minimal resource requirements and, hence, should be executed by only one or two processors. Since the range and hash partitioning strategies can decluster a relation only on a single attribute, both are forced to direct either the type A or the type B queries to all the processors, incurring the overhead of using more processors than absolutely necessary.

On the other hand, MAGIC declustering would construct a two dimensional directory on the Stock relation, as shown in Figure 4, in which each entry corresponds to a fragment - a disjoint subset of the tuples of the relation. The rows of the directory correspond to ranges of values for the P/E attribute, while the columns correspond to the intervals of the Symbol attribute value. The grid directory consists of 36

		Symbol					
		A-D	E-H	I-L	M-P	Q-T	U-Z
P/E	0-10	1	2	3	4	5	6
	11-20	7	8	9	10	11	12
	21-30	13	14	15	16	17	18
	31-40	19	20	21	22	23	24
	41-50	25	26	27	28	29	30
	51-∞	31	32	33	34	35	36

Figure 4: A two-dimensional declustering of Stock relation with MAGIC.

entries (i.e., fragments) and, assuming a system consisting of exactly 36 processors, each fragment will be assigned to a different processor (the details of how less contrived cases are handled is described in [36]). For example, tuples with Symbol attribute values ranging from letters A through D and P/E attribute values ranging from values 21 to 30 are assigned to processor 13.

Next, contrast the execution of queries A and B when the Stock table is hash partitioned on the Symbol attribute with when it is declustered using MAGIC and the assignment presented in Figure 4. Query type A is an exact match query on the Symbol attribute. The hash partitioning strategy localizes the execution of this query to a single processor. The MAGIC declustering strategy employs six processors to execute this query because its selection predicate maps to one column of the two dimensional directory. As an example, consider the query that selects the record corresponding to BMC Software (Stock.Symbol = "BMC"). The predicate of this query maps to the first column of the grid directory and processors 1, 7, 13, 19, 25, and 31 are employed to execute it.

Query type B is a range query on the P/E attribute. The hash partitioning strategy must direct this query to all 36 processors because P/E is not the partitioning attribute. Again, MAGIC directs this query to six processors since its predicate value maps to one row of the grid directory and the entries of each row have been assigned to six different processors. If instead the Stock relation was range partitioned on the P/E attribute, a single processor would have been used to execute the second query; however, then the first query would have been executed by all 36 processors.

Consequently, the MAGIC partitioning strategy uses an average of six processors, while the range and hash partitioning strategies both use an average of 18.5 processors. Ideally, however, a single processor should have been used for each query since they both have minimal resource requirements. Coming closer to using the optimal number of processors has two important benefits. First, the average response time of both queries is reduced because query initiation overhead [19] is reduced. Second, using fewer processors increases the overall throughput of the system since the "freed" processors can be used to execute additional queries.

3 Evaluation of the join operator using inter-operator parallelism

A common join operator is the equi-join operator, R.A = S.A. It concatenates a tuple of R with those tuples of S that have matching values for attribute A. This section describes sort-merge [26, 52, 53], Grace hash-join [73] and Hybrid hash-join [26, 86] to parallel process this operator. A common feature of these algorithms is their re-partitioning of relations R and S using the joining attribute A. This divides the join operator into a collection of disjoint joins that can be processed in parallel and independent of one another. Following a description of each algorithm, Section 3.1 describes how these techniques compare with one another and the role of multi-attribute partitioning strategies with these algorithms.

Sort-Merge

A parallel version of sort-merge join is a straightforward extension of its single-node implementation. Its details are as follows. First, the smaller of the two joining relations, R, is hash partitioned using attribute A. Its tuples are stored in a temporary files as they arrive at each node. Next, relation S is partitioned across the K nodes using the same hash function applied to attribute A. The use of the same hash function guarantees that those tuples of R at node 1 may join only with those of S at the same node. In a final step, a local merge join operation is performed by each node, in parallel with other nodes. The results might be stored in a file or pipelined onto other operators that might consume the result of this join operator.

Grace hash-join

The Grace hash-join algorithm [73] works in three steps. In the first step, the algorithm hash partitions relation R into N buckets using its join attribute A. In the second step, it partitions relation S into N buckets using the same hash function. In the last step, the algorithm processes each bucket B$_i$ of R and S to compute the joining tuples.

Ideally, N should be chosen in a manner so that each bucket is almost the same as the available memory without exceeding it. To accomplish this objective, the algorithm starts with a very large value for N. This reduces the probability of a bucket exceeding the memory size. If the buckets are much smaller than main memory, several will be combined during the third phase to approximate the available memory.

This algorithm is different than sort-merge in one fundamental way: In its last step, the tuples from bucket B$_i$ of R are stored in memory resident hash tables (using attribute A). The tuples from bucket B$_i$ of S are used to probe this hash table for matching tuples. Grace-join may use the smaller table (say R) to determine the number of buckets: this calculation is independent of the larger table (S).

Hybrid hash-join

The Hybrid hash-join also operates in three steps. Its main difference when compared with Grace hash-join is as follows. It maintains the tuples of the first bucket of R to build the memory resident hash table while constructing the remaining N-1 buckets are stored in temporary files. Relation S is partitioned using the same hash function. Again, the last N-1 buckets are stored in temporary files while the tuples in the first bucket are used to immediately probe the memory resident hash table for matching tuples.

3.1 Discussion

A comparison of sort-merge, Grace and Hybrid hash-join algorithms (along with other variants) is reported in [13, 25, 84]. In general, Grace and Hybrid provide significant savings when compared with sort-merge. Hybrid outperforms Grace as long as the first bucket does not overflow the available memory. Assuming that the size of R and S are fixed, both Hybrid and sort-merge are sensitive to the available memory size. Grace hash-join is relatively insensitive to the amount of available memory because it performs bucket tunning in the first step. The performance of Hybrid improves when large amounts of memory are available. Sort-merge also benefits (in a step-wise manner as a function of available memory) because it can sort R and S with fewer iterations of reading and writing each table.

One may employ bit filters [8, 91] to improve the performance of these algorithms. The concept is simple. An array of bits is initialized to zero. During the partitioning phase of R, a hash function is applied to the join attribute A of each tuple and the appropriate bit is set to one. The fully constructed bit filter is then used when partitioning relation S. When reading a record of S, the same hash function is applied to the joining attribute of each tuple. If the corresponding bit is checked then it is transmitted for further processing. Otherwise, there is no possibility of that tuple joining and it can be eliminated from further consideration. This minimizes the network traffic and subsequent processing, e.g., with sort-merge, the eliminated tuples are not sorted, reducing the number of I/Os.

One may control partitioning of tables to enhance the performance of the join operator. For example, the DYOP technique [77] distributes a data file into a set of partitions (or buckets) by repeatedly subdividing the tuple space of multiple attribute domains (in a fashion that is almost identical to the grid file algorithm). To execute a hash-join query efficiently, the size of each partition is defined to equal the aggregate memory of the processors in the system. Since the DYOP structure preserves the order of tuples in the attribute domain space, the bucket formation step of Grace hash-join algorithm is eliminated and the join of relations R and S is accomplished by reading each relation only once. Similarly, [61] also proposes the use of a multi-attribute partitioning to minimize the impact of data distribution during the construction of the hash table on the inner relation when executing a parallel hash join. The basic idea is as follows. Assume a relation R that is frequently joined with relations S and T. When R is joined with S, the A attribute of R is used and, when R is joined with T, the Y attribute of R is used as the joining attribute. By building a grid file on the A and Y attributes of R which is then used to decluster the tuples of R, it is possible to minimize how many tuples of R are redistributed when it is joined with either S or T.

4 ORE: A framework for data migration

While techniques such as MAGIC decluster a relation by analyzing its workload, this workload might evolve over time. Another challenge is the gradual evolution of a homogeneous system to a heterogeneous one. This might happen for several reasons. First, disks fail and it might be more economical to purchase newer disk models that are faster and cheaper than the original models. Second, the application might grow over time (in terms of both storage and bandwidth requirement) and demand additional nodes from the underlying hardware. Once again, it might be more economical to extend the existing configuration by purchasing newer hardware that is faster than the original nodes.

With evolving workloads and environments, data must be re-organized to respond to these changes. Ideally, the parallel DBMS should respond to these changes and fine-tune the placement of data. This can be performed at different granularities: 1) record level by repartitioning records and controlling assignment of records to each node [68], and 2) fragment level [83, 93, 37] by either migrating fragments from one node to another or breaking fragments into pieces and migrating some of its pieces to different nodes. We focus on the later approach in the rest of this chapter.

In order to simplify discussion and without loss of generality, we assume an environment consisting of K storage devices. In essence, each node of Figure 1.a is a storage device. Each storage device d_i has a fixed storage capacity, $C(d_i)$, an average bandwidth, $BW(d_i)$, and a Mean Time To Failure, $MTTF(d_i)$. With one or more applications that consume B_{total} bandwidth during a fixed amount of time, ideally, each disk must contribute a bandwidth proportional to its $BW(d_i)$:

$$Fairshare(d_i) = B_{total} \times \frac{BW(d_i)}{\sum_{i=1}^{K} BW(d_i)} \tag{1}$$

The bandwidth of a disk is a function of block size (β) and its physical characteristics [55, 9]: seek time, rotational latency, and transfer rate (tfr). It is defined as:

$$BW(d_i) = tfr \times \frac{\beta}{\beta + (tfr \times (seek\ time + rotational\ latency))} \tag{2}$$

Given a fixed seek time and rotational latency, BW(d_i) approaches disk transfer rate with larger block sizes.

There are F files stored on the underlying storage. The number of files might change over times, causing the value of F to change. A file f_i might be partitioned into two or more fragments. Its number of fragments

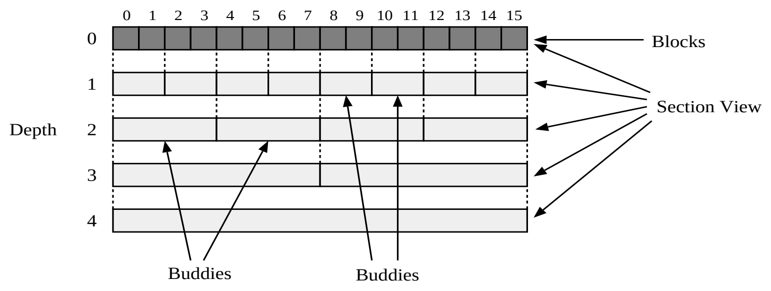


Figure 5: Physical division of disk space into blocks and the corresponding logical view of the sections with an example base of $B = 2$.

is independent of the number of storage devices, i.e., K. Fragments of a file may have different sizes. Fragment j of file f_i is denoted as $f_{i,j}$. In our assumed environment, two or more fragments of a file might be assigned to the same disk drive[1]. Moreover, a file f_i may specify a certain availability requirement from the underlying system. For example, it may specify that its Mean-Time-To-Data-Loss, $MTTDL(f_i)$, should exceed 200,000 hours, $MTTDL_{min}(f_i)$ = 200,000 hours.

We assume physical disk drives fail independent of one another. Each disk has a certain failure rate [103, 87, 47], termed $\lambda_{failure}$. Its mean-time-to-failure (MTTF) is simply:$\frac{1}{\lambda_{failure}}$. When a file (say f_j) is partitioned into n fragments and assigned to n disks (say d_1 to d_n) then the data becomes unavailable in the presence of a single failure[2]. Hence, it is defined as follows [103, 87, 47]:

$$MTTDL(f_i) = \frac{1}{\sum_{i=1}^{n} \lambda_{failure}(d_i)} \tag{3}$$

For example, if the MTTF of disk A and B is 1 million and 2 million hours, respectively, then the MTTDL of a file with fragments scattered across these two disks is 666,666 hours.

We use the EVEREST [40, 41] file system to approximate a contiguous layout of a file fragment on the disk drive. With EVEREST, the basic unit of allocation is a block, also termed sections of height 0. EVEREST combines these blocks in a tree-like fashion to form larger, contiguous sections. As illustrated in Figure 5, only sections of size(block) $\times B^i$ (for $i \geq 0$) are valid, where the base B is a system configuration parameter. If a section consists of B^i blocks then i is said to be the height of the section. In general, B height i sections (physically adjacent) might be combined to construct a height $i + 1$ section.

To illustrate, the disk in Figure 5 consists of 16 blocks. The system is configured with $B = 2$. Thus, the size of a section may vary from 1, 2, 4, 8, up to 16 blocks. In essence, a binary tree is imposed upon the sequence of blocks. The maximum height, given by[3] $N = \lceil \log_B(\lfloor \frac{Capacity}{\text{size(block)}} \rfloor) \rceil$, is 4. With this organization imposed upon the disk drive, sections of height $i \geq 0$ cannot start at just any block number, but only at offsets that are multiples of B^i. This restriction ensures that any section, with the exception of the one at height N, has a total of $B - 1$ adjacent *buddy* sections of the same size at all times. With the base 2 organization of Figure 5, each block has one buddy.

A fragment might be represented as several sections. Each is termed a *chunk*. The file system maintains the heat of each chunk at the granularity of a fixed offset from its section height. For example, with a chunk

[1]As compared with [83] that requires each fragment of a file to be assigned to a different disk drive.

[2]There has been a significant amount of research on construction of parity data blocks and redundant data, see [103] that focuses on this for heterogeneous disks. This topic is beyond the focus of this study. In this chapter, we control the placement without constructing redundant data.

[3]To simplify the discussion, assume that the total number of blocks is a power of B. The general case can be handled similarly and is described in [40, 41].

of height 8, the system might maintain its heat at offset 2. With B equal to 2, this means that the system maintains the heat of four section of height 6 that constitute this chunk. This enables the reorganization algorithm to break a fragment into many smaller pieces and disperse them amongst the available disk drives.

4.1 Three steps of ORE

Our framework consists of 3 logical steps: monitor, predict, and migrate. We partition time into fixed intervals, termed *time slices*. During **monitor**, we construct a profile of the load imposed by each file fragments per time slice. During **predict**, we compute what fragments to migrate from one disk to another in order to enhance system performance. **Migrate** changes the placement of candidate fragments. Below, we detail each of these steps.

4.1.1 Monitor

constructs a profile of the load imposed on each disk drive and the average response time of each disk d_i. The load imposed on disk drive d_i is quantified as the bandwidth required from disk d_i. It is the total number of bytes retrieved from d_i during a time slice divided by the duration of the time slice. The average response time of d_i is the average response time of the requests it processes during the time interval.

This process produces three tables that are used by the other two steps:

- FragProfiler table maintains the average block request size, heat, and load imposed by each fragment $f_{i,j}$ per time slice,
- For each disk drive d_i per time slice, DiskProfiler table maintains the heat, load, standard deviation in system load, average response time, average queue length, and utilization of d_i.
- FragOvlp table maintains the OVERLAP between two fragments per time slice. The concept of OVERLAP is detailed in Section 4.2.

4.1.2 Predict

determines what fragments to migrate to enhance response time. Section 4.2 describes several techniques that can be employed for this step. In Section 4.3, we quantify the tradeoff associated with these alternatives.

4.1.3 Migrate

modifies the placement of data. We considered two algorithms for fragment migration. With the first, the fragment is locked in exclusive mode while it is migrated from d_{src} to d_{dst}. This simple algorithm prevents updates while the fragment is migrating. It is efficient and easy to implement. However, the data might appear to be unavailable during the reorganization process. Due to this limitation, we ignore this algorithm from further consideration.

The second supports concurrent updates by performing each against two copies of the migrating fragment: (a) one on d_{src}, termed primary, and (b) the other on d_{dst}, termed secondary. The secondary copy is constructed from the primary copy of the fragment. All read requests are directed to the primary copy. All updates are performed against both the primary and secondary copy. The migration process is a background task that is performed based on availability of bandwidth from d_{src}. It assumes some buffer space for staging data from primary copy to facilitate construction of its secondary copy. This buffer space might be provided as a component of the embedded device. Depending on its size, the system might read and write units larger than a block. Moreover, it might perform writes against d_{dst} in the background depending on the amount of free buffer space. Once the free space falls below a certain threshold, the system might perform writes as foreground tasks that compete with active user requests [6].

4.2 Predict: Fragments to migrate

In this section, we describe two algorithms that strive to distribute the load of an application evenly across the K disks. These are termed EVEN and EVEN$_{C/B}$. As implied by their name, EVEN$_{C/B}$ is a variant of EVEN. A taxonomy of alternative techniques can be found in [37].

EVEN: Constrained by bandwidth

At the end of each time slice, EVEN computes the fair-share of system load for each disk drive. Next, it identifies the disk with (a) maximum positive load imbalance, termed d_{src}, and (b) minimum negative load imbalance, termed d_{dst}. (The concept of load imbalance is formalized in the next paragraph.) Amongst the fragments of d_{src}, it chooses the one with a load closest to the minimum negative load of d_{dst}. It migrates this fragment from d_{src} to d_{dst}. This process repeats until either there are no source and destination disks or a new time slice arrives.

The maximum positive load imbalance pertains to those disks with an imposed load greater than their fair share. For each such disk d_i, its $\delta^+(d_i)$ = load(d_i)-Fairshare(d_i). Positive imbalance of d_i is defined as $\frac{\delta^+(d_i)}{Fairshare(d_i)}$. EVEN identifies the disk with highest such value as the source disk, d_{src}, to migrate fragments from.

The minimum negative load imbalance corresponds to those disks with an imposed load less than their fair share. For each such disk d_i, its $\delta^-(d_i)$ = load(d_i)-Fairshare(d_i). Negative imbalance of d_i is $\frac{\delta^-(d_i)}{Fairshare(d_i)}$. The disk with the smallest negative imbalance[4] is the destination disk, d_{dst}, and EVEN migrates fragments to this disk.

EVEN defines XTRA as the difference between fair share of d_{src} and its current load, XTRA = $load(d_{src})$ - $Fairshare(d_{src})$. The difference between fair share of d_{dst} and its current load is termed LACKING, LACKING = $Fairshare(d_{dst})$ - $load(d_{dst})$. EVEN identifies fragments from d_{src} with an imposed load approximately the same as LACKING. Next, it migrates these fragments to d_{dst}.

EVEN$_{C/B}$: Constrained by bandwidth with Cost/Benefit Consideration

EVEN$_{C/B}$ extends EVEN by quantifying the benefit and cost of each candidate migration from d$_{src}$ to d_{dst}. The next paragraph describes how the system quantifies the cost and benefit of each candidate migration. EVEN$_{C/B}$ sorts candidate migration based on their net benefit, i.e., benefit - cost, scheduling the one that provides greatest savings. After each migration, the cost of each candidate migration is re-computed (because it might have changes) and the list is resorted. Section 4.3 shows that this algorithm outperforms EVEN.

In the rest of this section, we describe how to quantify the benefit and cost of migrating a fragment $f_{i,j}$ from d_{src} to d_{dst}. Its unit of measurement is time, i.e., milliseconds. The cost of migrating a fragment is the total time spent by d_{src} to read the fragment and d_{dst} to write the fragment.

The benefit of migrating $f_{i,j}$ is measured in the context of previous time slices. ORE hypothesizes a virtual state where $f_{i,j}$ resides on d_{dst} and measures the improvement in average response time. In essence, it estimates an answer to the following question: "What would be the average response time if $f_{i,j}$ resided on d_{dst}?" By comparing this with the observed response time, we quantify the benefit of a migration. Of course, this number might be a negative value which implies no benefit to performing this migration. Note that this methodology assumes that the past access patterns are an indication of future access patterns.

We start by describing a methodology to estimate an answer to the hypothetical "what-if" question. Next, we formalize how to compute the benefit.

Our methodology to estimate an answer to the "what-if" question is fairly accurate; its highest observed percentage of error is 23%. We realize this accuracy for two reasons: First, our embedded SAN file system resides in the SAN switch and observes all block references and the status of each storage device. Second, we maintain one additional piece of information, namely the degree of overlap between two fragments, termed

[4]Given two disks, d_1 and d_2 with negative imbalance of -0.5 and -2.0, respectively, d_2 has the minimum negative load imbalance.

OVERLAP($f_{i,j}$, $f_{k,l}$). This information is maintained for each time slice and used to predict response time.

In order to define OVERLAP and describe our methodology, and without loss of generality, assume that we are answering the "what-if" question in the context of one time slice. To simplify the discussion further, assume that the environment consists of homogeneous disk drives. (This assumption is removed at the end of this section.) The average system response time, RT_{avg}, is a function of average response time observed by requests referencing each fragment. Assuming F files, each partitioned into at most G fragments, it is defined as:

$$RT_{avg} \; = \frac{\sum_{i=1}^{F} \; \sum_{j=1}^{G} RT_{avg}(f_{i,j})}{F \; \times \; G} \tag{4}$$

The average response time of a fragment, $RT_{avg}(f_{i,j})$, is the sum of its average service time, $S_{avg}(f_{i,j})$, and wait time, $W_{avg}(f_{i,j})$, of requests that reference it:

$$RT_{avg}(f_{i,j}) \; = \; S_{avg}(f_{i,j}) \; + \; W_{avg}(f_{i,j}) \tag{5}$$

$S_{avg}(f_{i,j})$ is a function of the disk it resides on and average requested block size. For each fragment, as detailed in Section 4.1.1, ORE maintains the average requested block size in the FragProfiler table. Thus, given a disk drive d_{dst} and a fragment $f_{i,j}$, ORE can estimate $S_{avg}(f_{i,j})$ if $f_{i,j}$ resided on d_{dst} (using the physical characteristics of d_{dst}).

To compute W_{avg}, we note that each request has an arrival time, T_{arvl}, that can be registered by the embedded device. For each fragment $f_{i,j}$ residing on disk d_i, we maintain when the requests referencing $f_{i,j}$ will depart the system, termed T_{depart}. T_{depart} is estimated by analyzing the wait time in the queue of d_i. Upon the arrival of a request referencing fragment $f_{k,l}$, we examine all those fragments with a non-negative T_{depart}. For each, we set OVERLAP($f_{k,l}$, $f_{i,j}$,T_{arvl}) to be the difference between $T_{arvl}(f_{k,l})$ and $T_{depart}(f_{i,j})$: OVERLAP($f_{k,l}$, $f_{i,j}$,T_{arvl})= Max(0, $T_{depart}(f_{i,j}) \; - \; T_{arvl}(f_{k,l})$). For a time slice, OVERLAP($f_{k,l}$, $f_{i,j}$) is the sum of those OVERLAP($f_{k,l}$, $f_{i,j}$,T_{arvl}) whose T_{arvl} is during the time slice. In our implementation, we maintained OVERLAP($f_{k,l}$, $f_{i,j}$) as an integer that is initialized to zero at the beginning of each time slice. Upon the arrival of a request referencing $f_{k,l}$, we increment OVERLAP($f_{k,l}$, $f_{i,j}$) with OVERLAP($f_{k,l}$, $f_{i,j}$,T_{arvl}). This minimizes the amount of required memory.

OVERLAP($f_{k,l}$, $f_{i,j}$) defines how long requests referencing $f_{k,l}$ wait in a queue because of requests that reference $f_{i,j}$. Assuming that $f_{i,j}$ and $f_{k,l}$ are the only fragments assigned to disk d_i and the system processes #Req($f_{k,l}$) requests that reference $f_{k,l}$, the average wait time for these requests is:

$$W_{avg}(f_{k,l}) = \frac{OVERLAP(f_{k,l}, f_{i,j}) \; + \; OVERLAP(f_{k,l}, f_{k,l})}{\#Req(f_{k,l})} \tag{6}$$

It is important to observe the following two details. First, self OVERLAP is also defined for a fragment $f_{k,l}$, i.e., there exists a value for OVERLAP($f_{k,l}$, $f_{k,l}$). This enables ORE to estimate how long requests that reference the same fragment wait for one another. Second, this paradigm is flexible enough to enable ORE to maintain OVERLAP($f_{k,l}$, $f_{i,j}$) even when $f_{k,l}$ and $f_{i,j}$ reside on different disks. ORE uses this to estimate a response time for a hypothetical configuration where $f_{i,j}$ migrates to the disk containing $f_{k,l}$. Third, ORE can estimate the response time of a disk drive for an arbitrary assignment of fragments to disks using Equation 4.

Based on Equation 5, there are two ways to enhance response time observed by requests that reference a fragment. First, migrate the fragment to a faster disk for an improved service time, S_{avg}. Second, migrate a fragment $f_{i,j}$ away from those disks whose resident fragments have a high OVERLAP with $f_{i,j}$. Figure 6 shows the pseudo-code to estimate the benefit of migrating $f_{i,j}$ from d_{src} to d_{dst}. ORE may compute this for N previous time slices where N is an arbitrary number. The only requirement is that the embedded device must provide sufficent space to store all data pertaining to these intervals.

1. Number of accesses processed by disk d_{src} is $Access_{src}$
2. Number of accesses processed by disk d_{dst} is $Access_{dst}$
3. Look-up the average response time of d_{src} prior to migration, termed $RT_{src,before}$
4. Look-up the average response time of d_{dst} prior to migration, termed $RT_{dst,before}$
5. Estimate the average response time of d_{src} after migration, termed $RT_{src,after}$
6. Estimate the average response time of d_{dst} after migration, termed $RT_{dst,after}$
7. Total response time savings of d_{src} after migration is:
   $Savings_{src}$=$(Access_{src,after} \times RT_{src,after}) - (Access_{src,before} \times RT_{src,before})$.
8. Total response time savings of d_{dst} after migration is:
   $Savings_{dst}$=$(Access_{dst,after} \times RT_{dst,after}) - (Access_{dst,before} \times RT_{dst,before})$.
9. Benefit of migrating $f_{i,j}$ is $Benefit(f_{i,j})$=$Savings_{src} + Savings_{dst}$.

Figure 6: Pseudo-code to compute the benefit of a candidate migration.

The OVERLAP of two fragments is maintained in the FragOvlp table. Given G fragments, in the worst case scenario, the system maintains $\frac{G^2+G}{2}$ integer values. For example with a 630 fragments (G=630) and records that are 348 bytes long, in the worst case scenario, the system would store 65 megabytes of data per time slice. In our experiments, the amount of required storage was significantly less than this, only 70 kilobytes per time slice. With the 80-20 rule, we expect this to hold true for almost all applications. In Section 5, we describe how ORE can employ a circular buffer to limit the size of trace data that it gathers from the system.

4.3 Performance evaluation of ORE

We used a trace driven simulation study to quantify the performance of ORE. We analyzed two alternative environments: First, a homogeneous environment consisting of identical disk models. Second, a heterogeneous environment consisting of different disk models. For both environments, ORE provides significant performance enhancements. In the following, we start with a brief overview of the trace driven simulation model. Next, we present the obtained results for each environment and our observations.

The traces were gathered from a production Oracle database management system on a HP workstation configured with 4 gigabyte of memory, and 5 terabytes of storage devices (283 raw devices). The database consisted of 70 tables and is 27 gigabyte in size. The traces were gathered from 4 pm, April 12 to 1 pm April 23, 2001. It corresponds to 23 million operations on the data blocks. The file reference is skewed where approximately 83% of accesses reference 10% of the files. Moreover, accesses to the tables are bursty as a function of time. This is demonstrated in Figure 7, where we plot the number of requests to the system as a function of time. In all experiments, the duration of a time slice is 6 minutes, i.e., each tick on the x-axis of the presented figures is 6 minutes long.

We used the Java programming language to implement our simulation model. It consists of 3 class definitions:

1. Disk: This class definition simulates a multi-zone disk drive with a complete analytical model for computing seeks, rotational latency, and transfer time. When a disk object is instantiated, it reads its system parameters from a database management system. Hence, we can configure the model with different disk models and different number of disks for each model. A disk implements a simplified version of the EVEREST file system.

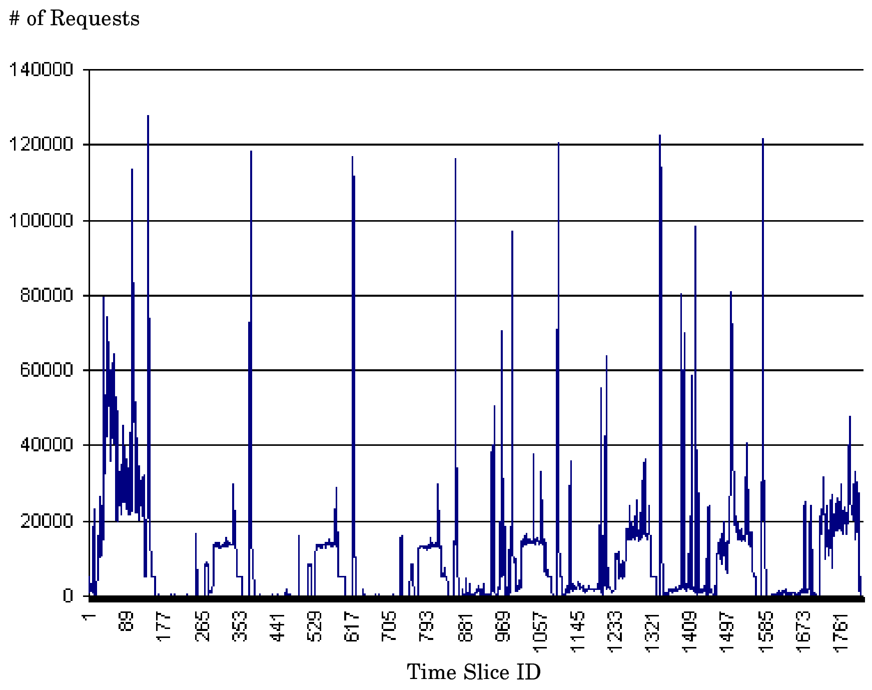


Figure 7: Number of requests as a function of time.

2. Client: The client generates requests for the different blocks by reading the entries in the trace files.

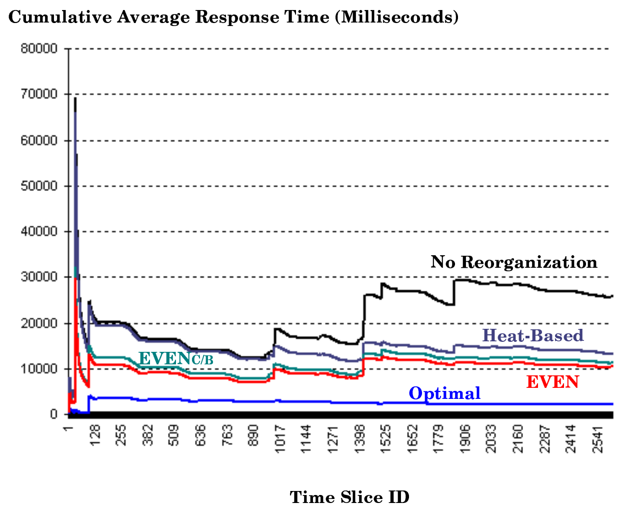
3. SAN Switch: This class definition implements a simplified SAN switch that routes messages between the client and the disk drives. The file manager is a component of this module. The file manager services each request generated by a client. It controls and maintains the placement of data across disk drives. Given a request for a block of a file, this module locates the fragment referenced by the request and resolves which disk contains the referenced data. It consults with the file system of the disk drive to identify the appropriate cylinder and track that contains the referenced block.

The file manager implements the 3-step re-organization algorithm of ORE, see Section 4.

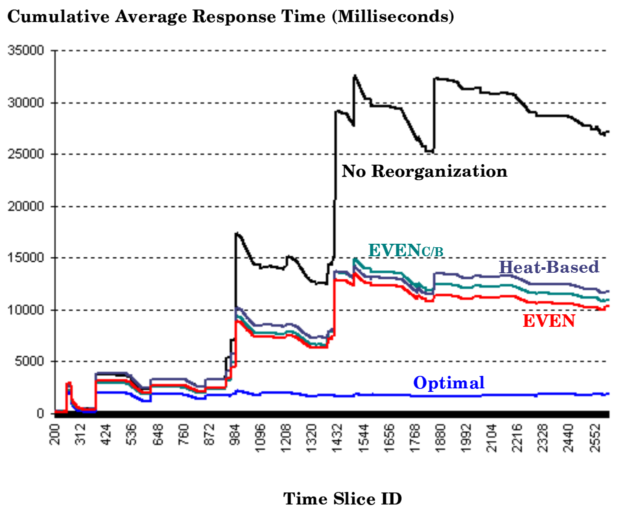
We conducted experiments with both a large configuration consisting of 283 raw devices that corresponds to the physical system that produced the traces and smaller configurations. The smaller configurations are faster to simulate. The performance results presented in this chapter are based on one such configuration consisting of 9 disk drives. We analyze two environments: First, a homogeneous one consisting of nine 180 gigabyte disk drives with a transfer rate of 40 megabytes per second (MB/sec). These disks were modelled after the high density, Ultra160 SCSI/Fibre-Channel disks introduced by Seagate in late 2000. Our second environment is a heterogeneous one consisting of three different disk models: 1) three disk drives identical to those used with the homogeneous environment, 2) three 60 gigabyte disk drives, each with a transfer rate of 20 MB/sec, and 3) three 20 gigabyte disk drives, each with a transfer rate of 4 MB/sec.

4.4 Homogeneous configuration

Figure 8 shows the performance of alternative predict techniques using the trace. The x-axis of this figure denotes time, i.e., different time slices. The y-axis is the cumulative average response time. It is computed as follows. For each time slice, we compute the total number of requests and the sum of all response times



8a. Starting with time slice 1



8b. Starting with time slice 200

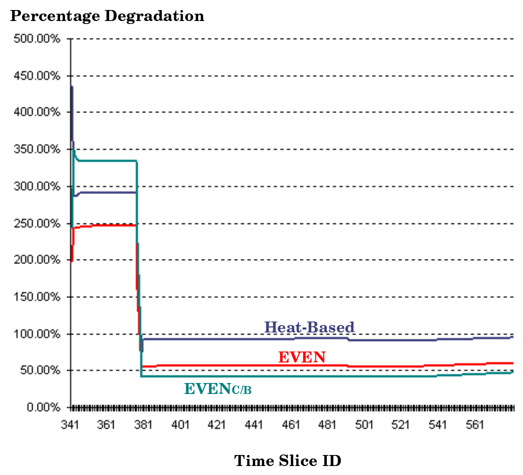
Figure 8: Cumulative average response time for the homogeneous environment

til the end of that time slice. The cumulative average response time is the ratio of these two numbers, i.e., $\frac{total\ response\ time}{total\ requests}$. If during a time slice, no requests are issued then the cumulative average response time remains constant. This explains the periodic flat portions.

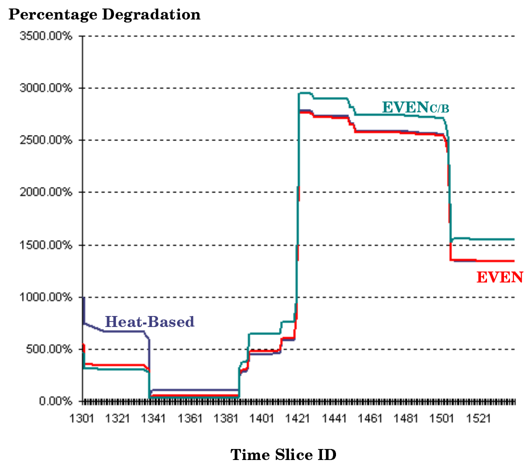
In addition to EVEN and EVEN$_{C/B}$, these figures present the response time for three other configurations. These correspond to:

- No-reorganization: this represents the base configuration that processes requests without on-line re-organization.
- Optimal: this configuration assigns requests to the disks in a round-robin manner, ignoring the placement of data and files referenced by each request. This configuration represents the *theoretical* lower bound on response time that can be obtained from the 9 disk configuration.
- Heat-Based: This is an implementation of the re-organization algorithm presented in [83]. Briefly, this algorithm monitors the *heat* [19] of disks and migrates the fragment with highest temperature from the hottest disk to the coldest one if: a) the heat of the target disk after this migration does not exceed the heat of the source disk, and b) the hottest disk does not have a queue of pending requests. The heat of a fragment is defined as the sum of the number of block accesses to the fragment per time unit, as computed using statistical observation during some period of time. The temperature of a fragment is the ratio between its heat and size. The heat of a disk is the sum of the heat of its assigned fragments [19, 62].

Figure 8.a and b show the cumulative average response time starting with the 1th and 200th time slice, respectively. The former represents a cold start while the later is a warm start after 20 hours of using the framework. In both cases, ORE is a significant improvement when compared with no-reorganization. (ORE refers to the framework consisting of the three possible algorithms: EVEN, EVEN$_{C/B}$, and Heat-Based.) The peaks in this figure correspond to the bursty arrival of requests which result in the formation of queues. Even though Optimal assigns requests to the nodes in a round-robin manner, it also observes formation of queues because many requests arrive in a short span of time.



9.a Day 2



9.b Day 6

Figure 9: Percentage degradation relative to Optimal

We also analyzed the performance of alternative algorithms on a daily basis. This was done as follows. We set the cumulative average response time to zero at midnight on each day. In all cases, ORE outperforms no re-organization. When compared with the theoretical Optimal, ORE is slower by an order of magnitude. Figure 9 shows how inferior EVEN, EVEN$_{C/B}$ and Heat-Based are when compard with Optimal. The y-axis on this figure is the percentage difference between an algorithm (say EVEN) and Optimal. A large percentage difference is undesirable because it is further away from the ideal. We show two different days, corresponding to the best and worst observed performance. During day 2, ORE is 50 to 300 percent slower than the theoretical Optimal. During day 6, ORE is at times several orders of magnitude slower than Optimal.

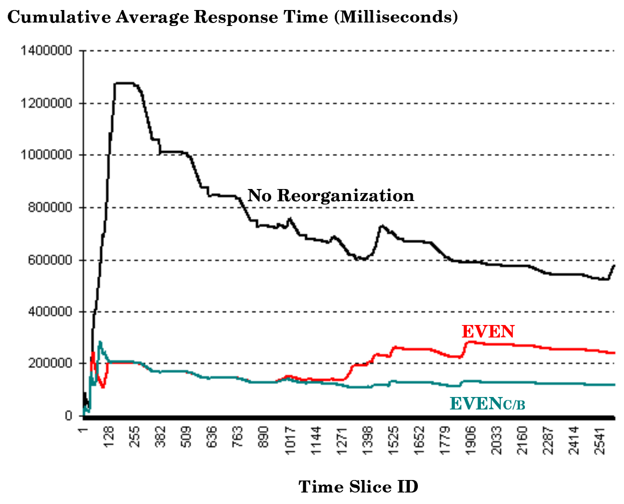
4.5 Heterogeneous configuration

We analyzed the performance of a heterogeneous configuration consisting of 9 disk drives. These disks correspond to 3 different disk models: 1) 180 gigabyte disks with a 40 megabyte per second transfer rate, 2) 60 gigabyte disks with a 20 megabyte per second transfer rate, and 3) 20 gigabyte disks with a 4 megabyte per second transfer rate. Our environment consisted of 3 disks for each model.

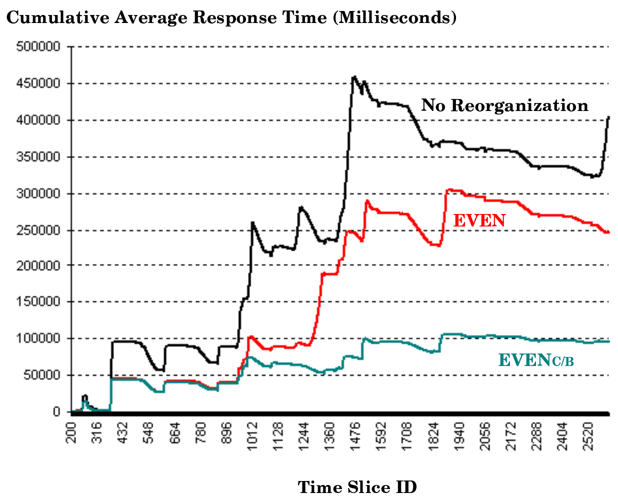
The experimental setup differed from that of Section 4.4 in several ways. First, we increased the block size to 128 kilobyte. With a 2 kilobyte block size, the bandwidth of each disk is almost identical because the seek and rotational delays constitute more than 99% of transfer time, see Equation 2. Second, we do not have Optimal because, with a heterogeneous configuration, the faster disks can service requests faster and it is no longer optimal to assign the requests to disks in a round-robin manner. Similarly, we eliminated the Heat-Based technique because its extension to a heterogeneous environment would be similar to EVEN.

Figure 10 shows the cumulative average response time of the system with EVEN, EVEN$_{C/B}$, and no-reorganization. These results demonstrate the superiority of ORE as a re-organization framework. EVEN$_{C/B}$ enhances performance for several reasons. First, it migrates the fragments with a high imposed load to the faster disks, processing a larger fraction of requests faster. Thus, when a burst of requests is issued to these fragments, each request spends less time in the queue. Second, it migrates the fragments that are referenced together onto different disk drives in order to minimize the incurred wait time (using the concept of OVERLAP).

We compared EVEN with EVEN$_{C/B}$ on a day-to-day basis. This procedure is identical to those of the homogeneous configuration where the cumulative average response time is reset to zero at the begining of



10a. Starting with time slice 1



10b. Starting with time slice 200

Figure 10: Cumulative average response time for the heterogeneous environment

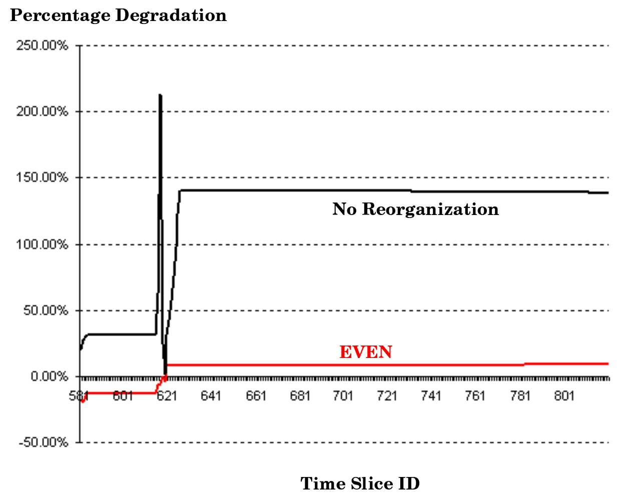
each day, 12 am. Generally speaking, EVEN$_{C/B}$ is superior to EVEN. In Figures 11.a and b, we show the percentage degradataion relative to EVEN$_{C/B}$ observed for two different days, day 3 and 6. These correspond to the best and worst observed performance with EVEN. During day 3, EVEN provides a performance that is at times better than EVEN$_{C/B}$. During day 6, EVEN exhibits a performance degradation that is several orders of magnitude slower than EVEN$_{C/B}$. In this case, no re-organization outperforms EVEN.

5 Conclusions and future research directions

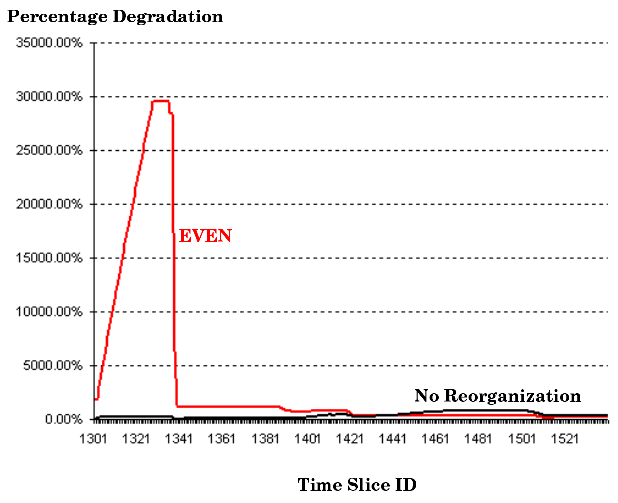
This chapter provides an overview of techniques to realize a parallel, scalable, high performance database management system. We described the role of alternative partitioning strategies to distribute the workload of a query across multiple nodes. Next, we described the design of parallel sort-merge, Grace and Hybrid hash join to process the join operator. Finally, we detailed ORE as a three step framework that controls the placement of fragments in order to respond to: a) changing workloads, and b) dynamic hardware platforms that evolve over a period of time. We demonstrated the superiority of this framework using a trace driven evaluation study.

Physical design of parallel database management systems is an active area of research. One emerging complementary effort is in the area of Storage Area Network (SAN) architectures that strive to minimize the cost of managing storage. A SAN is a special-purpose network that interconnects different data storage devices with servers. While there are many definitions for a SAN, there is a general consensus that it provides access at the granularity of a block and is typically targeted toward database applications. A SAN might include an embedded storage management software in support of virtualization. This software includes a file system that separates storage of a device from the physical device, i.e., physical data independence. Virtualization is important because it enables a file to grow beyond the capacity of one disk (or disk array). Such embedded file systems can benefit from ORE and its 3 step framework [37].

Another important research direction is an online capacity planner that is aware of an application's performance requirements, e.g., desired response time guarantees at pre-specified throughputs. This component should detect when a system is not meeting the desired requirements and suggest changes to the hardware platform. With a SAN, this might be an integral component of the embedded file system. Such a capacity



11a. Day 3



11b. Day 6

Figure 11: Percentage degradation relative to EVEN$_{C/B}$

planner empowers the human operators to address performance limitations effectively.

Finally, we plan to extend ORE to incorporate availability [78, 103] techniques. These techniques construct redundant data in order to continue operation in the presence of disk failures. For example, chain declustering [59, 60, 49] constructs a backup copy of a fragment assigned to node 1 onto an adjacent node 2. The original fragments on node 1 are termed *primary* while their backup copies on node 2 are termed *secondary*. If node 1 fails, the system continues operation using secondary copies stored on node 2. While ORE, see Section 4, controls the placement of data based on the availability needs of a fragment, it does not consider the placement of primary and secondary copies when migrating fragments from one node to another. As a simple example, it can switch the role of primary and backup copies to respond to workload changes.

6 Acknowledgments

We wish to thank Anouar Jamoussi and Sandra Knight of BMC Software for collecting and providing traces used in this study. We also thank William Wang, Sivakumar Sethuraman, and Dinakar Yanamandala of USC for assisting with the implementation of our simulation model.

References

[1] Y. Amir, A. Peterson, and D. Shaw. Seamlessly Selecting the Best Copy from Internet-Wide Replicated Web Servers. In *International Symposium on Distributed Computing*, 1998.

[2] K. Amiri, G. Gibson, and R. Golding. Highly Concurrent Shared Storage. In *Proceedings of the International Conference on Distributed Computing Systems*, April 2000.

[3] K. Amiri, D. Petrou, G. Ganger, and G. Gibson. Dynamice Function Placement for Data-Intensive Cluster Ccomputing. In *Proceedings of the USENIX Annual Technical Conference*, June 2000.

[4] T. Anderson, Y. Breitbart, H. Korth, and A. Wool. Replication, consistency, and practicality: Are these mutually exclusive? *Proceedings of ACM Special Interest Group on Management of Data*, 27, 1998.

[5] W. Aref, I. Kamel, and S. Ghandeharizadeh. Disk Scheduling in Video Editing Systems. *To appear in IEEE Transactions on Knowledge and Data Engineering*, 2002.

[6] W. Aref, I. Kamel, T. Niranjan, and S. Ghandeharizadeh. Disk Scheduling for Displaying and Recording Video in Non-Linear News Editing Systems. In *Proceedings of Multimedia Computing and Networking Conference*, 1997.

[7] R. H. Arpaci-Dusseau, E. Anderson, N. Treuhaft, D. E. Culler, J. M. Hellerstein, D. Patterson, and K. Yelick. Cluster I/O with River: Making the fast case common. In *Proceedings of the Sixth Workshop on Input/Output in Parallel and Distributed Systems*, pages 10–22, Atlanta, GA, 1999. ACM Press.

[8] E. Babb. Implementing a relational database by means of specialized hardware. *ACM Transactions on Database Systems*, 4(1):1–29, 1979.

[9] S. Berson, S. Ghandeharizadeh, R. Muntz, and X. Ju. Staggered Striping in Multimedia Information Systems. In *Proceedings of ACM Special Interest Group on Management of Data*, pages 79–90, 1994.

[10] R. Bhatia, R. K. Sinha, and C. Chen. Declustering using golden ratio sequences. In *ICDE*, pages 271–280, 2000.

[11] P. Biswas, K. Ramaakrishnan, and D. Towsley. Exploiting Non-Volatile Memory in Disks for Write Caching. In *Proceedings of ACM SIGMETRICS*, 1993.

[12] H. Boral, W. Alexander, L. Clay, G. Copeland, S. Danforth, M. Franklin, B. Hart, M. Smith, and P. Valduriez. Prototyping Bubba, a highly parallel database system. *IEEE Transactions on Knowledge and Data Engineering*, 2(1), 1990.

[13] K. Bratbergsengen. Hashing Methods and Relational Algebra Operations. In *Proceedings of the Very Large Databases Conference*, pages 323–333, 1984.

[14] S. Chaudhuri, S. Ghandeharizadeh, and C. Shahabi. Retrieval of Composite Multimedia Objects. In *Proceedings of the VLDB Conference*, September 1995.

[15] L-T. Chen and D. Rotem. Declustering objects for visualization. In *Proceedings of the Very Large Databases Conference*, 1993.

[16] P. M. Chen, G. A. Gibson, R. H. Katz, and D. A. Patterson. An Evaluation of Redundant Arrays of Disks Using an Amdahl 5890. In *Measurement and Modeling of Computer Systems*, pages 74–85, 1990.

[17] P. M. Chen and D. A. Patterson. Maximizing Performance in a Striped Disk Array. In *Proc. 17th Annual Int'l Symp. on Computer Architecture, ACM SIGARCH Computer Architecture News*, page 322, 1990.

[18] H. Chou, D. J. DeWitt, R. Katz, and T. Klug. Design and implementation of the Wisconsin Storage System. *Software Practices and Experience*, 15(3), 1985.

[19] G. Copeland, W. Alexander, E. Boughter, and T. Keller. Data placement in Bubba. In *Proceedings of ACM Special Interest Group on Management of Data*, pages 99–108, 1988.

[20] G. Copeland, W. Alexander, E. Boughter, and T. Keller. Data Placement in Bubba. In *Proceedings of ACM Special Interest Group on Management of Data*, pages 99–108, 1988.

[21] O. P. Damani, P. E. Chung, Y. Huang, C. Kintala, and Y. Wang. ONE-IP: Techniques for hosting a service on a cluster of machines. *Computer Networks and ISDN Systems*, 29(8–13):1019–1027, 1997.

[22] A. Dan and D. Sitaram. An Online Video Placement Policy Based on Bandwidth to Space Ratio (BSR). In *Proceedings of ACM Special Interest Group on Management of Data*, pages 376–385, 1995.

[23] D. DeWitt, S. Ghandeharizadeh, D. Schneider, A. Bricker, H. Hsiao, and R. Rasmussen. The Gamma Database Machine Project. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):44–62, March 1990.

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

[25] D. J. DeWitt and Gerber.R. Multiprocessor hash-based join algorithms. In *Proceedings of the Very Large Databases Conference*, 1985.

[26] D. J. DeWitt, R. H. Katz, F. Olken, L. D. Shapiro, M. R. Stonebraker, and D. Wood. Implementation techniques for main memory database systems. *ACM Special Interest Group on Management of Data Record*, 14(2):1–8, 1984.

[27] A. L. Drapeau, K. W. Shirrif, J. H. Hartman, E. L. Miller, S. Seshan, R. H. Katz, K. Lutz, D. A. Patterson, E. K. Lee, P. H. Chen, and G. A. Gibson. RAID-II: A high-bandwidth network file server. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 234–244, 1994.

[28] H. C. Du and J. S. Sobolewski. Disk allocation for Cartesian product files on multiple-disk systems. *ACM Transactions on Database Systems*, 7(1):82–101, 1982.

[29] J. Eisenstein, S. Ghandeharizadeh, L. Huang, C. Shahabi, G. Shanbhag, and R. Zimmermamnn. Analysis of Clustering Techniques to Detect Hand Signs. In *Proceedings of the International Symposium on Intelligent Multimedia, Video and Speech Processing*, 2001.

[30] J. Eisenstein, S. Ghandeharizadeh, C. Shahabi, G. Shanbhag, , and R. Zimmermann. Alternative representations and abstractions for moving sensors databases. In *Proceedings of the ACM Tenth International Conference on Information and Knowledge Management (CIKM'01)*, November 2001.

[31] C. K. Baru et al. DB2 Parallel Edition. *IBM Systems Journal*, 34(2):292–322, 1995.

[32] C. Faloutsos and P. Bhagwat. Declustering using fractals. In *PDIS*, pages 18–25, 1993.

[33] C. Faloutsos and D. Metaxas. Declustering using error correcting codes. In *Proc. Symp. on Principles of Database Systems*, pages 253–258, 1989.

[34] S. Ghandeharizadeh, V. Choi, C. Ker, and K. Lin. Design and Implementation of the Omega Object-based System. In *Proceedings of the fourth Australian Database Conference*, February 1993.

[35] S. Ghandeharizadeh and D. DeWitt. A Multiuser Performance Aanalysis of Alternative Declustering Strategies. In *Proceedings of the 6th IEEE Data Engineering Conference*, 1990.

[36] S. Ghandeharizadeh and D. J. DeWitt. MAGIC: a multiattribute declustering mechanism for multiprocessor database machines. *IEEE Transactions on Parallel and Distributed Systems*, 5(5):509–524, 1994.

[37] S. Ghandeharizadeh, S. Gao, C. Gahagan, and R. Krauss. An On-Line Reorganization Framework for Embedded SAN File Systems. In *Submitted for publication*, 2001.

[38] S. Ghandeharizadeh, L. Hua, and I. Kamel. A Novel Cost Driven Disk Scheduling Algorithm for Multi-Priority Multimedia Objects. *To appear in IEEE Transactions on Multimedia*, 2002.

[39] S. Ghandeharizadeh, D. Ierardi, and D. Kim. Placement of Data in Multi Zone Disk Drives. In *Proceedings of the Second International Baltic Workshop on Databases and Information Systems*, 1996.

[40] S. Ghandeharizadeh, D. Ierardi, and R. Zimmermann. An Algorithm for Disk Space Management to Minimize Seeks. *Information Processing Letters*, 57:75–81, 1996.

[41] S. Ghandeharizadeh, D. Ierardi, and R. Zimmermann. Management of Space in Hierarchical Storage Systems. In M. Arbib and J. Grethe, editors, *A Guide to Neuroinformatics.* Academic Press, 2001.

[42] S. Ghandeharizadeh, S. Kim, W. Shi, and R. Zimmermann. On minimizing startup latency in scalable continuous media servers. In *Multimedia Computing and Networking*, Feb 1997.

[43] S. Ghandeharizadeh, S. Ho Kim, and C. Shahabi. On Configuring a Single Disk Continuous Media Server. In *Proceedings of ACM SIGMETRICS*, pages 37–46, 1995.

[44] S. Ghandeharizadeh and R. Muntz. Design and Implementation of Scalable Continuous Media Servers. *Parallel Computing*, pages 91–122, 1998.

[45] S. Ghandeharizadeh, R. Zimmermann, W. Shi, R. Rejaie, D. Ierardi, and T. Li. Mitra: A Scalable Continuous Media Server. *Multimedia Tools and Applications*, 5(1):79–108, 1997.

[46] S. Ghanderharizadeh and D. DeWitt. A performance analysis of alternative multi-attribute declustering strategies. In *Proceedings of ACM Special Interest Group on Management of Data*, pages 29–38, June 1992.

[47] G. Gibson. Redundant Disk Arrays: Reliable, Parallel Secondary Storage, 1991.

[48] G. A. Gibson and D. A. Patterson. Designing disk arrays for high data reliability. *Journal of Parallel and Distributed Computing*, 17(1–2):4–27, /1993.

[49] L. Golubchik and R. R. Muntz. Fault Tolerance Issues in Data Declustering for Parallel Database Systems. *Data Engineering Bulletin*, 17(3):14–28, 1994.

[50] M. E. Gomez and V. Santonja. Analysis of Self-Similarity in I/O Workload Using Structural Modelling. In *Proceedings of the 7th International Symposim on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, 1998.

[51] V. Gottemukkala and E. Omiecinski. The sensible sharing appriach to a scalable, high-performance database system. Technical Report GIT-CC-93-24, Georgia Institute of Technology, March 1993.

[52] G. Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys*, 25(2):73–170, 1993.

[53] G. Graefe. Sort-Merge-Join: An Idea Whose Time has Passed? In *Proc. IEEE Conf. on Data Engineering*, pages 406–417, 1994.

[54] G. Graefe. Volcano - An Extensible and Parallel Query Evaluation System. *IEEE Transactions on Knowledge and Data Engineering*, 6(1), 1994.

[55] J. Gray and G. Graefe. The 5 Minute Rule, Ten Years Later. In *ACM Special Interest Group on Management of Data Record*, volume 26, 1997.

[56] J. Gray, P. Helland, P. O'Neil, and D. Shasha. The dangers of replication and a solution. In *Proceedings of ACM Special Interest Group on Management of Data*, pages 173–182, 1996.

[57] J. Gray, B. Horst, and M. Walker. Parity Striping of Disk Arrays: Low Cost Reliable Storage with Acceptable Throughput. In *Proceedings of the Very Large Databases Conference*, pages 152–162, September 1990.

[58] J. Gray and A. Reuter. *Transaction Processing : Concepts and Techniques.* Morgan Kaufmann, 1992.

[59] H. Hsiao and D. DeWitt. Chained Declustering: A New Availability Strategy for Multiprocessor Database Machines. In *Proceedings of 6th International Data Engineering Conference*, pages 456–465, 1990.

[60] Hui-I Hsiao and David DeWitt. A performance study of three high availability data replication strategies. In *Proceedings of the First International Conference on Parallel and Distributed Information Systems*, pages 18–28, 1991.

[61] K. Hua and C. Lee. An Adaptive Data Placement Scheme for Parallel Database Computer Systems. In *Proceedings of the Very Large Databases Conference*, 1990.

[62] Randy H. Katz and Wei Hong. The performance of disk arrays in shared-memory database machines. *Distributed and Parallel Databases*, 1(2):167–198, 1993.

[63] Myoung Ho Kim and Sakti Pramanik. Optimal file distribution for partial match retrieval. In *Proceedings of ACM Special Interest Group on Management of Data*, pages 173–182, 1988.

[64] L. Kleinrock. *Queuing Systems.* Wiley Interscience, 1975.

[65] T. Kroeger and D. Long. Predicting File System Actions from Prior Events. In *Proceedings of the USENIX*, 1996.

[66] M. K. Lakhamraju, R. Rastogi, S. Seshadri, and S. Sudarshan. On-Line Reorganization in Object Databases. In *Proceedings of ACM Special Interest Group on Management of Data*, pages 58–69, 2000.

[67] E. K. Lee and C. A. Thekkath. Petal: Distributed virtual disks. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 84–92, Cambridge, MA, 1996.

[68] M. L. Lee, M. Kitsuregawa, B. C. Ooi, K. Tan, and A. Mondal. Towards self-tuning data placement in parallel database systems. In *Proceedings of ACM Special Interest Group on Management of Data*, pages 225–236, 2000.

[69] J. Li, J. Srivastava, and D. Rotem. CMD: A multidimensional declustering method for parallel data systems. In *Proceedings of the Very Large Databases Conference*, 1992.

[70] M. Livny, S. Khoshafian, and H. Boral. Multi-disk management algorithms. In *Proceedings of the 1987 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 69–77, 1987.

[71] M. McKusick, W. Joy, S. Leffler, and R. Fabry. A Fast File System for UNIX. *ACM Transactions on Computer Systems*, 2(3):181–197, August 1984.

[72] B. Moon and J. Saltz. Scalability Analysis of Declustering Methods for Multidimensional Range Queries. *IEEE Transactions on Knowledge and Data Engineering*, 10(2):310–327, March 1998.

[73] M. Nakano, M. Kitsuregawa, and M. Takagi. Query Execution for Large Relation on Functional Disk System.". In *Proc. IEEE CS Intl. Conf. No. 5 on Data Engineering, Los Angeles*, 1989.

[74] Oracle & Digital. *Oracle Parallel Server in Digital Environment*, 1994. Technical Report, Oracle Inc.

[75] S. Ortiz. Embedded OSs Gain the Inside Track. *IEEE Computer*, 34(11):14–16, 2001.

[76] E. Ottem and R. Gundersen. High Availability Clusters and Storage Area Networks. Technical report, Gadzoox Inc, 2001.

[77] E. Ozkarahan and M. Ouksel. Dynamic and Order Preserving Data Partitioning for Database Machines. In *Proceedings of the Very Large Databases Conference*, 1985.

[78] D. Patterson, G. Gibson, and R. Katz. A Case for Redundant Arrays of Inexpensive Disks (RAID). In *Proceedings of ACM Special Interest Group on Management of Data*, pages 109–116, 1988.

[79] D. Petrou, K. Amiri, G. Ganger, and G. Gibson. Easing the Management of Data-Parallel Systems via Adaptation. In *Proceedings of the 9th ACM SIGOPS European Workshop*, September 2000.

[80] V. J. Ribeiro, R. H. Riedi, M. S. Crouse, and R. G. Baraniuk. Simulation of nonGaussian Long-Range-Dependent Traffic Using Wavelets. In *Proceedings of ACM SIGMETRICS*, pages 1–12, 1999.

[81] D. Ries and R. Epstein. Evaluation of Distribution Criteria for Distributed Database Systems. Technical Report UCB/ERL Technical Report M78/22, UC Berkeley, May 1978.

[82] P. Scheuermann, G. Weikum, and P. Zabback. "Disk Cooling" in Parallel Disk Systems. *Data Engineering Bulletin*, 17(3):29–40, 1994.

[83] P. Scheuermann, G. Weikum, and P. Zabbak. Data Partitioning and Load Balancing in Parallel Disk Systems. *Very Large Databases Journal*, 7(1), 1998.

[84] D. A. Schneider and D. J. DeWitt. A performance evaluation of four parallel join algorithms in a shared-nothing multiprocessor environment. In *Proceedings of ACM Special Interest Group on Management of Data*, pages 110–121, 1989.

[85] T. Schroeder, S. Goddard, and B. Ramamurthy. Scalable Web Sever Clustering Technologies. *IEEE Network*, pages 38–45, May 2000.

[86] L. Shapiro. Join Processing in Database Systems with Large Main Memories. *ACM Transactions on Database Systems*, 11(3), 1986.

[87] D. P. Siewiorek and R. S. Swarz. *The Theory and Practice of Reliable System Design.* Digital Press, 1982.

[88] M. Stonebraker, R. Katz, D. Patterson, and J. Ousterhout. The design of XPRS. In *Proceedings of the Very Large Databases Conference*, 1988.

[89] Tandem Performance Group. A benchmark of non-stop SQL on the debit credit transaction. In *Proceedings of ACM Special Interest Group on Management of Data*, pages 337–341, 1988.

[90] Teradata Corp. *DBC/1012 Data Base Computer System Manual*, November 1985. Teredata Corp. Document No. C10-0001-02, Release 2.0.

[91] P. Valduriez. Join and Semi-Join Algorithms for a Multiprocessor Database Machine. *ACM Transactions on Database Systems,* 9(2), March 1984.

[92] A. Veitch, E. Riedel, S. Towers, and J. Wilkes. Towards global storage management and data placement. Technical Report HPL-SSP-2001-1, Hewlett Packard Laboratories, March 2001.

[93] R. Vingralek, Y. Breitbart, and G. Weikum. Snowball: Scalable storage on networks of workstations with balanced load. *Distributed and Parallel Databases*, 6(2):117–156, 1998.

[94] W. Vogels, D. Dimitriu, K. Birman, R. Gamache, R. Short, J. Vert, J. Barrera, and J. Gray. The Design and Architecture of the Microsoft Cluster Service. In *IEEE FTCS*, 1998.

[95] M. Wang, T. Madhyastha, N. H. Chan, S. Papadimitriou, and C. Faloutsos. Data Mining Meeting Performance Evaluation: Fast Algorithms for Modeling Bursty Traffic. Technical report, Carnegie Mellon University, 2001.

[96] G. Weikum, C. Hasse, A. Moenkeberg, and P. Zabback. The COMFORT automatic tuning project, invited project review. *Information Systems*, 19(5):381–432, 1994.

[97] Gerhard Weikum. The Web in 2010: Challenges and Opportunities for Database Research. In *Informatics*, pages 1–23, 2001.

[98] M. Wiesmann, F. Pedone, A. Schiper, B. Kemme, and G. Alonso. Understanding replication in databases and distributed systems. In *Proceedings of 20th International Conference on Distributed Computing Systems (ICDCS'2000)*, pages 264–274, Taipei, Taiwan, R.O.C., 2000. IEEE Computer Society Technical Commitee on Distributed Processing.

[99] J. Wilkes, R. Golding, C. Staelin, and T. Sullivan. The HP AutoRAID Hierarchical Storage System. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, pages 96–108, Copper Mountain, CO, 1995. ACM Press.

[100] K. Wu, P. Yu, J. Chung, and J. Teng. A Performance Study of Workfile Disk Management for Concurrent Mergesorts in a Multiprocessor Database System. In *Proceedings of the Very Large Databases Conference*, pages 100–110, September 1995.

[101] X. Yu, B. Gum, Y. Chen, R. Wang, K. Li, A. Krishnamurthy, and T. Anderson. Trading Capacity for Performance in a Disk Array. In *Symposium on Operating Systems Design and Implementation*, October 2000.

[102] E. Zegura, M. Ammar, Z. Fei, and S. Bhattacharjee. Application Layer Anycasting: A Server Seclection Architecture and Use in a Replicated Web Service. *ACM/IEEE Transactions on Networking*, 8(4):455–466, 2000.

[103] R. Zimmermann and S. Ghandeharizadeh. HERA: Heterogeneous Extension of RAID. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA 2000)*, June 2000.