PERFORMANCE AND AVAILABILITY IN DATABASE MACHINES WITH REPLICATED DATA

by

Hui-I Hsiao

Computer Sciences Technical Report #963
August 1990

PERFORMANCE AND AVAILABILITY IN DATABASE MACHINES WITH REPLICATED DATA

by

HUI-I HSIAO

A thesis submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy
(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN — MADISON

1990

ABSTRACT

In this thesis, we present a new strategy for increasing the availability of data in the context of multiprocessor, shared-nothing database machines. This technique, termed **chained declustering**, is demonstrated to provide superior performance in the event of failures while maintaining a very high degree of data availability. Furthermore, unlike most earlier replication strategies, the implementation of chained declustering requires no special hardware and only minimal modifications to existing software.

To better understand the relative performance of different data replication strategies in various situations, we have implemented a simulation model of the Gamma database machine. Using this simulation model, we explore the performance of alternative replication strategies under different workload assumptions. Among the issues that we have examined are (1) the relative performance of different strategies when no failures have occurred, (2) the effect of a single node failure on system throughput and response time, (3) the performance impact of varying the CPU speed and/or disk page size on the different replication strategies, and (4) the tradeoff between the benefit of intra query parallelism and the overhead of activating and scheduling extra operator processes.

The simulation results indicate that, in the event of a disk failure, chained declustering provides noticeably better performance than interleaved declustering and much better performance than mirrored disks. When a failure has not occurred, chained declustering can potentially provide better performance than mirrored disks for I/O bound applications, and performance comparable to mirrored disks for CPU bound applications. In addition, in the normal mode of operation, chained declustering provides performance comparable to interleaved declustering for read only applications and provides slightly higher throughput than interleaved declustering for applications that require both disk reads and writes.

ACKNOWLEDGMENTS

It has been my privilege and good fortune to work and study with Professor David DeWitt for whom I have the greatest respect. David has provided excellent guidance, continued encouragement, generous support, and the opportunity to attend national and international database conference. His enthusiasm, encouragement, and patience have made this thesis possible.

I am very grateful to Professor Michael Carey for his guidance, extremely helpful comments and suggestions, especially during the last year of my graduate study. I would also like to thank Professor Miron Livny for providing numerous suggestions to improve the presentation of this dissertation and providing the DeNet simulation environment used for conducting this research. Thanks also go to the other members of my committee, Professors Jeffrey Naughton and Gurindar Sohi, for their comments and suggestions.

Finally, my special thanks to my parents, my wife Hui-Shin, my son Daniel, and my daughter Emily for their understanding, patience, and support during my graduate career.

This research was partially supported by the Defense Advanced Research Projects Agency under contract N00039-86-C-0578, by the National Science Foundation under grant DCR-8512862, by a Digital Equipment Corporation External Research Grant, and by a research grant from the Tandem Computer Corporation.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

1.1. Introduction and Motivation

While a number of solutions have been proposed for increasing the availability and reliability of computer systems, the most commonly used technique involves the replication of processors and mass storage [Borr81, Jone83, Kast83]. Some systems go one step further, replicating not only hardware components, but also software modules, so that when a hardware or software module fails, the redundant hardware or software modules can continue running application software [Borr81]. The result is that application programs are isolated from almost all forms of failures.

For database applications, the availability of disk-resident data files is perhaps the major concern. Most database management systems employ a log and checkpointing of the database to ensure integrity and availability of the database in the event of disk or system failures. The log and checkpointing technique, however, can not satisfy the high data availability requirements of some current database applications[1]. This is because the recovery time in the case of a media failure is long, and during the recovery period data is unavailable.

To provide high data availability, two basic approaches have been proposed and used in database applications: the identical copy approach and the disk array approach (or error-correcting code approach). The identical copy approach stores multiple copies (usually two) of the same data item on different disks attached to separate processors. When one copy fails, the other copy can continue to be used. Unless both copies fail at the same time, the failure of a single copy will be transparent to users and no interruption of service will occur. Existing high availability strategies based on this approach include mirrored disks [Katz78, Bitt88], interleaved declustering [Tera85], and the inverted file strategy [Cope88].

[1] Some examples of such applications are stock market trading, air defense systems, air traffic control systems, airline reservation-type systems, banking (OLTP), etc.

Instead of storing two (or more) identical copies, the disk array approach uses an array of disks and stores the data and the redundant error detection/correction information (usually parity bytes) on different disk drives. When errors are discovered, the redundant information can be used to restore the data and application programs can continue using the data with minimal interruption. Strategies based on this approach include synchronized disk interleaving [Kim86], redundant array of inexpensive disks (RAID) [Patt88], and recently parity striping of disk arrays [Gray90].

Both approaches have been used in commercial systems. For example, Tandem's NonStop SQL database machine and Teradata's DBC 1012 database machine adopt the identical copy approach, while IBM's AS400 system is based on the disk array approach. Then, which of the above two approaches is a better choice for database applications? The tradeoffs between these two approaches can be captured by the following three factors: performance, availability, and cost.

While, traditionally, the performance of a computer system is measured in terms of both response time and throughput, in a multiprocessor system that provides resiliency from hardware and software failures, performance can be measured in two different operating modes: the **normal mode**, with no failed components[2], and the **failure mode**, in which one or more processors or disks have failed. In terms of the normal mode operation, a study by Gray [Gray90] has shown that mirrored disks (an identical copy based strategy) provides better performance than RAID for OLTP applications. In a separate study, Chen, Patterson et. al. [Chen90] demonstrate that for small requests (transferring less than one track of data per request), mirrored disks provides higher throughput (MB/s/disk) than RAID. Because an I/O request in most database applications is likely to transfer less than one track of data, the identical copy approach is potentially more suitable for database applications than the disk array approach. In terms of operations in the failure mode, systems based on the identical copy approach again have the potential to provide better performance than those based on the disk array approach. The reason is the following. With the identical copy approach (storing multiple copies), the remaining copy or copies can continue to be used when one copy fails. There will be little or no service degradation to users and/or application programs. On the other hand, with a disk array, when a query needs to access data on the failed disk/copy, the data would have to be reconstructed on the fly. This means that all remaining disks in the failed array must be accessed[3] in order to satisfy

[2] For the remainder of this dissertation, we will focus our attention only on hardware failures and will ignore software failures.

a single disk request. In such cases, the failed disk array will be restricted to serve only one request at a time and its performance will degrade significantly.

The availability of a system or data file is greatly influenced by the way data files are placed on disks[4] and the time needed to recover or restore a failed disk/node. The longer the recovery time is, the higher the probability will be of second failure resulting in data being unavailable. With the identical copy approach, data can be copied from the intact disk(s) to the new disk when the failed disk is replaced or fixed. This process can be done easily and quickly. As a result, the vulnerable window where a second failure may result in unavailability of data is short. With the disk array approach, on the other hand, all disk pages in the failed array must be read and processed in order to rebuild the data originally stored on the failed drive. This process will take longer than simply copying from one disk to the other. Consequently, when failures occur, systems employing a disk array will remain in the failure mode longer and the possibility of second failure occuring before the first failure is fixed will be higher than with systems employing the identical copy approach. As a result, the probability of data being unavailable with the disk array approach is higher than with the identical copy approach.

On the other hand, the disk array approach is more attractive if disk space and/or total system cost is a major concern. The reason is that a disk array stores less redundant information; the size of redundant error checking and recovery information normally is only a fraction of the stored data, while the size of the second copy is identical to the first copy with the identical copy approach.

Since performance and availability are/will be among the more important factors in future DBMSs, and since the cost of disk space is getting lower and accounts for a relatively small fraction of the total system expense, we believe that the identical copy approach is a better choice for applications that require high data availability and high performance (even though the cost will be somewhat higher). In this dissertation, we will focus our attention on strategies based on the identical copy approach and will only briefly discuss strategies based on the disk array approach.

---

[3] In addition, pages read from all disks would have to be "xor'd" together to reconstruct the failed data.

[4] The impact of data placement on data availability will be discussed in Chapter 4.

1.2. A New Availability Strategy

Throughout this dissertation, we focus our attention on multiprocessor database machines that employ a "shared-nothing" architecture [Ston86]. In such systems, each processor has its own main memory, processors communicate through an interconnection network, and one or more disks are connected to each processor. For multiprocessor, shared-nothing database machines with replicated data, the application of horizontal partitioning (i.e. declustering) techniques [Ries78, Tera85, DeWi86, Livn87] facilitates the successful application of inter and intra query parallelism in the normal mode of operation [DeWi86, Livn87, Tand87, DeWi88]. However, when a failure occurs, balancing the workload among the remaining processors and disks can become difficult, as one or more nodes[5] (processor/disk pairs) must pick up the workload of the component that has failed. In particular, unless the data placement scheme used allows the workload of the failed node to be distributed among the remaining operational nodes, the system will become unbalanced and the response time for a query may degrade significantly even though only one, out of perhaps a thousand nodes, has failed. In addition, the overall throughput of the system may be drastically reduced since a bottleneck may form.

In this dissertation we present a new declustering technique, termed **chained declustering**, for managing replicated data. This new technique provides high availability and, in addition, is able to fully balance[6] the workload among the operational nodes in the event of a failure. To balance the workload in the failure mode, a special load balancing algorithm has been designed in conjunction with the chained declustering scheme. This algorithm predetermines the **active fragments** of the primary and backup copies of each relation and directs data accesses to these fragments in a manner that will fully balance the workload in both modes of operation. The active fragments are initially designated at the time of database creation and are dynamically reassigned when node failures occur.

To better understand the relative performance of different data replication strategies in various situations, we have implemented a simulation model of the Gamma database machine. Using this simulation model, we have evaluated the performance of three data replication strategies, chained declustering, mirrored disks, and interleaved declustering, under a number of different workload assumptions. Among the issues that we have examined are (1) the relative performance of different strategies when no failures have occurred, (2) the effect of a single node failure

[5] We assume that, in the absence of special purpose hardware (i.e. dual ported disks and disk controllers), the failure of a processor controlling one or more disks renders the data on these disks unavailable.

[6] Assume that a relation is declustered over N disks. We call a data replication method **fully balanced** if the workload can be evenly distributed over N-i disks when i disks fail.

on system throughput and response time, and (3) the performance impact of varying the CPU speed and/or disk page size on the different replication strategies, and (4) the tradeoff between the benefit of intra query parallelism and the overhead of activating and scheduling extra operator processes.

1.3. Organization of the Dissertation

The organization of the rest of the dissertation is as follows. In the next chapter, related availability strategies based on both the identical copy and disk array approaches are presented. The chained declustering strategy is described in Chapter 3 along with the associated data placement algorithms. The load balancing algorithm designed specifically for use with the chained declustering strategy is also described in Chapter 3. Chapter 4 contains a comparison of the availability and performance of the chained declustering strategy and the existing availability strategies through an analytical study. In this chapter, we assume that failures are independent and that failure rates are exponentially distributed. We then study the degree of availability that each of the strategies is able to provide. We also compare the ability of each availability strategy to balance the workload among the remaining disks/nodes in the event of a disk/processor failure.

Our simulation model is described in Chapter 5. The performance results of our simulation experiments are presented and analyzed in Chapter 6. As one will see, our results indicate that chained declustering can provide superior performance in both the normal and failure modes of operation. Concurrency control and failure recovery issues associated with the chained declustering strategy are discussed in Chapter 7. Our conclusions and future research directions are contained in Chapter 8.

CHAPTER 2

AN OVERVIEW OF RELATED HIGH AVAILABILITY STRATIGIES

2.1. Introduction

In this chapter, we describe several existing strategies for improving data availability. The strategies discussed here include: the use of mirrored disks [Borr81], interleaved declustering [Tera85], inverted files [Cope88], synchronized disk interleaving (SDI) [Kim86], and disk arrays with redundant error recovery information (RAID) [Patt88]. The mirrored disk and interleaved declustering strategies employed by Tandem and Teradata, respectively, both maintain two identical copies of each relation. The inverted file strategy proposed in the Bubba database machine project [Cope88] also employs two copies of each relation but stores the second copy as a combination of inverted indices over the first copy plus a remainder file containing the uninverted attributes. RAID and SDI employ error correcting techniques that can be used to rebuild the database in the event of a disk failure.

Each of these strategies is able to sustain a single disk failure and thus provides resiliency to disk failures. With the Bubba, Tandem, and Teradata schemes, the remaining copy can continue to be used when the primary copy fails, while in RAID and SDI, data on the remaining disks can be accessed and manipulated to satisfy data requests for data stored on the failed drive. In the following, we will first describe strategies based on the identical copy approach, then the inverted file strategy, and finally we describe the disk array strategy.

2.2. Tandem's Mirrored Disks Architecture

The hardware structure of a Tandem system [Borr81] consists of one or more clusters that are linked together by a token ring. Each cluster contains 2 to 16 processors with multiple disk drives. The processors within a cluster are interconnected with a dual high speed (~20 Mbyte/sec) bus. Each processor has its own power supply, main memory, and I/O channel. Each disk drive is connected to two I/O controllers, and each I/O controller is connected to two processors, providing two completely independent paths to each disk drive. Furthermore, each disk drive is "mirrored" (duplicated) to further insure data availability.

Relations in Tandem's NonStop SQL [Tand87] system are generally declustered across multiple disk drives. For example, Figure 2.1 shows relation R partitioned across four disks using the mirrored disk strategy. R_i represents the i-th horizontal fragment of the first copy of R and r_i stands for the mirror image of R_i. As shown in Figure 2.1, the contents of disks 1 and 2 (and 3 and 4) are identical.

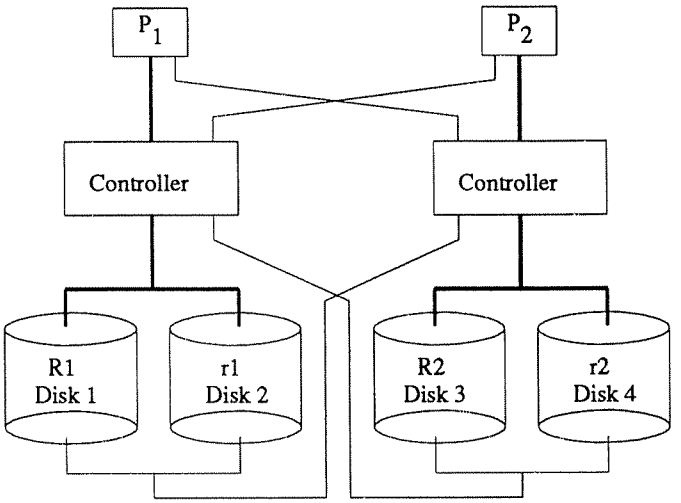

Figure 2.1: Data Placement with Tandem's Mirrored Disk Scheme.

Disks are accessed by a mechanism known as I/O "process-pairs". A process pair consists of two cooperating processes which execute on the two processors physically connected to a specific mirrored-disk pair. One of the processes, designated the "primary" process, controls the disk and handles all I/O requests to it. The other process serves as "backup" to the primary process and is activated should a failure occur in the primary process. Read operations can be directed (by the I/O controller) to either drive in the mirrored-pair while write operations must be directed to both drives in order to keep the contents of both disks identical, causing the two disk arms to become synchronized on writes [Bitt88].

When a disk in a mirrored pair fails, the remaining disk can assume the workload of the failed drive. Unless both disks in a mirrored pair fail at the same time, data will always be available. The actual impact of a drive failure on the performance of the system depends on the fraction of read and write operations [Bitt88]. If most I/Os are read operations, losing a drive may result in doubling the average I/O time because only one disk arm is available. On the other hand, if most I/Os are write operations, the impact of a disk failure may be minimal.

The impact of the failure of a processor will, however, almost always have a significant negative impact on performance. Consider the failure of processor P1 in Figure 2.1. While the data on disks 1 and 2 will remain

available (because the mirrored pair is dual ported), processor P2 will have to handle **all** accesses to disks 1 and 2 as well as disks 3 and 4 until processor P1 is repaired. If P2 is already fully utilized when the failure occurs, the response time for queries that need to access data on either pair of drives may double if the system is CPU bound.

2.3. Teradata's Interleaved Declustering Scheme

In the Teradata database machine [Tera85], the processors (and the disk drives attached to them) are subdivided into clusters. Each cluster contains from 2 to 16 processors and each processor may have one or two disk drives.[1] Relations are declustered among the disk drives within one or more clusters by hashing on a "key" attribute. Optionally, each relation can be replicated for increased availability. In this case, one copy is designated as the primary copy and the other as the backup or fallback copy.

The tuples of a relation stored on each disk are termed a **fragment**. Fragments of the primary and backup copies are termed primary and backup fragments respectively. Each primary fragment is stored on one node. For backup fragments, Teradata employs a special data placement scheme termed **interleaved declustering** [Tera85, Cope89]. If the cluster size is N, each backup fragment will be subdivided into N-1 subfragments. Each of these subfragments will be stored on a different disk within the same cluster other than the disk containing the primary fragment. Figure 2.2 illustrates the interleaved declustering mechanism. In Figure 2.2, a relation R is declustered across 8 disk drives. (R_i represents the i-th primary fragment whereas r_i represents the i-th backup fragment.)

	cluster 0				cluster 1			
Node	0	1	2	3	4	5	6	7
Primary Copy	R0	R1	R2	R3	R4	R5	R6	R7
Backup Copy		r0.0	r0.1	r0.2		r4.0	r4.1	r4.2
	r1.2		r1.0	r1.1	r5.2		r5.0	r5.1
	r2.1	r2.2		r2.0	r6.1	r6.2		r6.0
	r3.0	r3.1	r3.2		r7.0	r7.1	r7.2	

Figure 2.2: Interleaved Declustering (Cluster Size N = 4)

When a node failure occurs, this strategy is able to do a better job of balancing the load than the mirrored disk scheme since the workload of the failed node will be distributed among N-1 nodes instead of a single node. However, this improvement in load balancing is not without cost. As we will demonstrate in Chapter 4, the probability

[1] Actually, if two disk drives are used, they are treated as one logical drive.

of data being unavailable increases proportionately with the size of the cluster.[2]

During the normal mode of operation, read requests are directed to the fragments of the primary copy and write operations result in both copies being updated. In the event of a node failure that renders a fragment of the primary copy unavailable, the corresponding fragment of the backup copy will be promoted to become the primary (active) fragment and all data accesses will be directed to it.

2.4. Bubba's Data Placement Method

Another data replication scheme, termed the **inverted-file** (IF) technique, was used by the Bubba database machine [Cope88]. Unlike the two previous schemes, this scheme does not store two identical copies of a logical relation. Instead, one copy is a base relation termed the **direct copy** (D copy), and another copy is a set of inverted files and a remainder relation containing all non-inverted attributes, which together form what is termed the **IF** copy.

To enhance availability, the D copy and the IF copy of a relation are distributed over two **disjoint** sets of disks. The number of disks occupied by each of these two copies is determined by the following formulas:

$$B_D = \frac{Heat_{Dcopy}}{Heat_{Dcopy} + Heat_{IFcopy}} * C\_DegDecl,$$

$$B_{IF} = C\_DegDecl - B_D,$$

where C_DegDecl is the number of disks (C_DegDecl ≤ total number of disks) available for storing the tuples of the relation and the *heat* of an object(D or IF copy) is the access frequency of that object over some period of time. If the D copy and IF copy of a relation, together, are partitioned across all disk drives (C_DegDecl = total number of disks) and queries exhibit a uniform access pattern[3] over time, then C_DegDecl will equal the total number of disks and B_D and B_{IF} will each be a fixed number.

Depending on the number of inverted files, recovery from media or node failure will require different amounts of work. According to the designers, an advantage of the IF scheme is that it can process exact match selection queries very efficiently. Conversely, a drawback of this scheme is that the time to recover from a failure will be considerably longer than with the identical copy based schemes; one must either rebuild the inverted files from the direct copy or merge (by sorting and comparing the pointers) all the inverted files and remainder file to

[2] Data will be unavailable if any two nodes in a cluster fail.

[3] Bubba proposes to reorganize the data placement dynamically based on access frequency. With a uniform access pattern, no reorganization will be needed. Also, the past access pattern does not guarantee the future access pattern and the reorganization is not a simple task.

recreate the direct copy. Either alternative is more time consuming than simply copying one file to another. As a consequence, when failures occur, Bubba will remain in failure mode longer than systems that employ an identical copy based scheme. If failure-handling performance is the major concern, the Bubba scheme will thus not work as well as the identical copy based schemes. On the other hand, if the disk storage utilization is more important, then this scheme may be able to save some disk space over the identical copy based schemes since inverted files are needed anyway. However, as we will demonstrate below, there are several other strategies that use less disk space than the Bubba scheme and are still able to provide a high degree of availability.

2.5. Synchronized Disk Interleaving (SDI)

An alternative to using data replication to achieve a higher degree of data availability is termed synchronized disk interleaving (SDI) [Kim 86]. Instead of replicating data on different nodes, this strategy stores *check bytes* and *parity bytes* for recovering errors from both random single byte failures and single disk failures. The basic idea of this scheme is to interleave (decluster) each data block across multiple disk drives such that data in a block can be accessed in parallel. A group of disks is interleaved if each data block is stored in such a way that succeeding portions of the block are on different disks. For example, assume a data block X is 80K bytes long and there are 10 disks with a 4K byte sector size. If the interleaving granularity is a disk sector, the first 4K bytes will be stored on the first disk, the second 4K bytes on the second disk, ..., the eleventh 4K bytes on the first disk, and so on. When processing block X, all 10 disks can be accessed in parallel and the rate of data transfer can be increased by a factor of 10. Figure 2.3 illustrates an example of data placement in SDI. In Figure 2.3, the unit of interleaving is a physical disk sector. S_i represents the i-th sector of the stored data block (the i-th 4K bytes of block X in the example above) and C_i is the check byte sector for sub-block i. All check sectors are stored on the same disk (disk n+1 in the figure). The elements in one row form a sub-block which is used as the unit of error checking and recovery. The elements in each column are stored on one disk such that elements in different columns can be read/written in parallel.

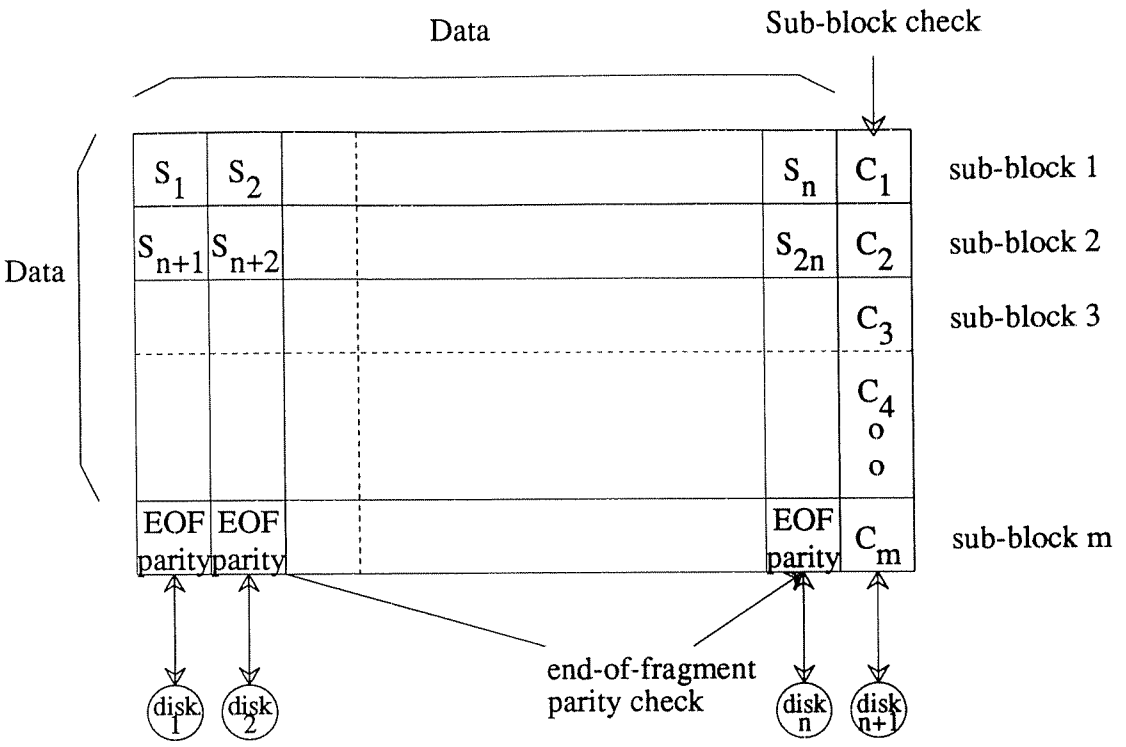

Figure 2.3: Data Placement in Synchronized Disk Interleaving.

In this scheme, the unit of interleaving is arbitrary: it can be as small as a single bit or as large as a physical disk sector. The redundant information consists of parity bytes and check bytes that are used for error checking and recovering; the end-of-fragment parity bytes are used to determine the location of failures while the sub-block check bytes are used for both error checking and recovery. The error of a single byte can be recovered by "xor"ing all remaining good data bytes (one byte from each data disk) in the same sub-block (same row) and comparing the parity bit-by-bit with the check byte stored in the check disk. If they match, the failed bit is a 0 otherwise it is a 1. Reconstructing the data of a failed disk can be done in the same fashion as a single byte recovery. Refer to [Kim 86] for a detailed description of the algorithm.

With this scheme, all accesses to data blocks are processed using a conversion buffer that resides either in main memory or in the disk controller. For read operations, this conversion buffer is used to merge data from different disks to form a block. In the case of writes, it is used to split a block into N fragments and each fragment is sent to a single disk.

This strategy provides both high bandwidth for large block transfers and higher data availability. It has been used in supercomputer systems such as Cray and has significantly improved the performance of some applications such as reading/writing very large data arrays. The drawback is that only one I/O per interleaved group can be performed at a time if a sector is not used as the unit of interleaving. This is because the information of an individual transfer unit (sector) is spread across several disks. As a result, this scheme will waste rather than gain I/O bandwidth for data transfers smaller than one sub-block. Even if a disk sector is used as an interleaving unit, there will still be an I/O bottleneck in the system because a write to any data disk will also require updating the check bytes on the check disk. For queries that perform frequent disk updates (e.g. order entry and debit-credit), this method will thus perform poorly. As will be seen in the next section, an enhanced version of this scheme alleviates this problem and increases the applicability of the scheme to database applications.

2.6. RAID's Data Storage Scheme

In the RAID data storage scheme [Patt88, Gibs89], an array of small, inexpensive disks is used as a single storage unit termed a *group*. Instead of replicating data on different drives, this strategy stores *check (parity) bytes* for recovering from both random single byte failures and single disk failures. This scheme interleaves (stripes) each data block, in units of one disk sector, across multiple disk drives such that all the sectors in a single block can be accessed in parallel, resulting in a significant increase in the data transfer rate.

Figure 2.4 illustrates an example of data placement in a RAID[4] group with five disks. $S_{i,j}$ represents the j-th sector of the i-th data block and C_i is the check byte sector for block i. Basically, C_i is formed by computing the "xor" (byte wise) of all data sectors in the block. As illustrated by the shaded sectors in Figure 2.4, the check byte sectors in a RAID group are distributed evenly among the disk drives.

[4] [Patt88] presents 5 levels of RAID. The scheme described here is the level 5 RAID. Except for the end-of-fragment parity byte, level 3 RAID is the same as the SDI [Kim86].

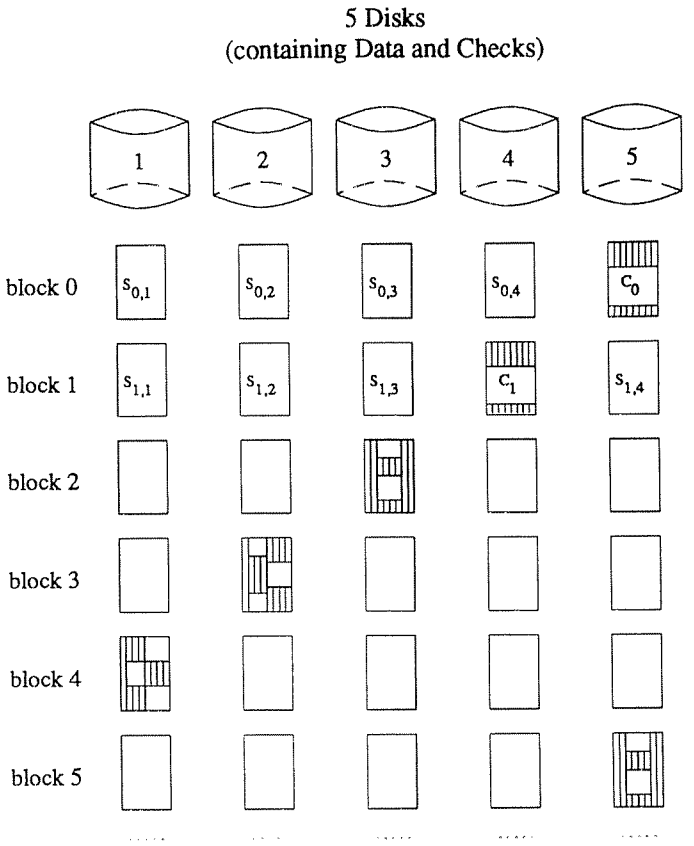

Figure 2.4: Data placement in RAID, each rectangle represents a physical sector.

If no disks have failed, reading one sector of a block requires access to only a single disk. Writing a sector, on the other hand, requires 4 disk accesses: a read and write of the sector being updated plus a read and write of the check sector for the corresponding block. When a disk failure occurs, however, every disk in the same RAID group must be accessed each time an access is attempted to a sector on inoperative disk drive. Recovering the value of each byte of a sector from the failed drive is similar to that with SDI. First, the corresponding bytes from the other sectors in the block (except the check sector) are "xor'd" with each other. The resulting byte is compared bitwise with the corresponding byte from the check sector for the block to determine whether the failed bit is a 0 (if the two bits match), or it is a 1 (if the two bits don't match).

One potential concern with the RAID scheme is its inherent reliability. When large numbers of disks are connected to form a single system, the increased number of disks and support hardware (cables, controllers, etc.) will increase the probability of a component failure which in turn increases the probability of unavailable data. This concern is supported by the results in [Schu88], where for a 56-disk RAID system, the probability of data being una-

vailable as the result of a non-media failure was shown to be as much as a factor of 30 higher than if only media failures are considered.

CHAPTER 3

CHAINED DECLUSTERING - A NEW AVAILABILITY STRATEGY

3.1. Introduction

While each of the techniques presented in the previous chapter significantly improves the probability of data remaining available in the event of a failure, each has one or more significant limitations. While the use of mirrored disk drives offers the highest level of availability (as will be shown in Chapter 4), it does a poor job of distributing the load of a failed node. Teradata's scheme provides a tradeoff between availability and performance in the event of a failure. As the cluster size is increased, the probability of two failures rendering data unavailable increases while the imbalance in workloads among the processors in the event of a failure decreases. The "disk array" strategies (RAID, SDI, etc.) emphasize the importance of disk space over performance.

In this chapter, we describe a new data replication technique, termed **chained declustering** which offers both high availability and excellent load balancing in the event of a failure. Chained declustering, being an "identical copy" based scheme, does however require more disk space than the "disk array" based strategies. In the following discussion, we assume that each processor has only a single disk attached and the term "node" is used to represent a processor-disk pair.

3.2. Data Placement with Chained Declustering

With chained declustering, two physical copies of each relation, termed the **primary copy** and the **backup copy**, are maintained. The tuples in the primary and backup copies of a relation are declustered over a set of disks and the primary and backup copies of the same fragment are guaranteed to be placed on different nodes. Unlike other identical copy strategies, the backup copy in chained declustering is used not only as a backup, but also as a tool for load balancing when node failures occur.

Nodes are divided into disjoint groups called **relation-clusters** and tuples of each relation are declustered among the drives that form one of the relation-clusters. The disks in each **relation-cluster** can be sub-divided into smaller groups termed **chain-clusters**. A small system may contain only one **relation-cluster** (i.e., the **relation-**

cluster has the same number of disks as the system has), while a large system may contain several. Similarly, each relation-cluster can contain one or more **chain-clusters**. In the following discussion, we first assume that the **relation-cluster** consists of only one **chain-cluster** which contains all the disks in the system. The generalized case is described in Section 3.3.

The data placement algorithm for chained declustering operates as follows. Assume that there are a total of M disks numbered from 1 to M. For every relation R, the i-th primary fragment is stored on the

{[i+C(R)] mod M}-th disk,

and the i-th backup fragment is stored on the

{[i+1+C(R)] mod M}-th[1] disk.

The function C(R) is devised to allow the first fragment of relation R to be placed on any disk within a **relation-cluster** while the "1" in the second formula is used to ensure that the identical fragments from the same relation will be placed on different disks. As an example, consider Figure 3.1 where M, the number of disks in the relation-cluster, is equal to 8 and C(R) is 0. The tuples in the primary copy of relation R are declustered using one of Gamma's three horizontal partitioning strategies with tuples in the i-th primary fragment (designated R_i) stored on the i-th disk drive. The backup copy is declustered using the same partitioning strategy but the i-th backup fragment (designated r_i) is stored on (i+1)-th disk (except r7 which is stored on 0-th disk). In the figure, R_i and r_i contain identical data. We term this data replication method **chained declustering** because the disks are "linked" together, by the fragments of a relation, like a chain.

Node	0	1	2	3	4	5	6	7
Primary Copy	R0	R1	R2	R3	R4	R5	R6	R7
Backup Copy	r7	r0	r1	r2	r3	r4	r5	r6

Figure 3.1: Chained Declustering (Relation Cluster Size = 8)

3.3. The Generalized Data Placement Algorithm

For systems having relatively few disks, it is reasonable to let the relation-cluster contain all of the disks in the system. However, for systems with a large number of disks (e.g. 1000) it will generally not be practical to decluster a relation over all of the drives as the overhead of initiating and committing transactions will generally overshadow

[1] A generalized formula is {[i+k+C(R)] mod M} where 0 < k < M and the greatest common divisor of M and k, GCD(M, k), is equal to 1.

the benefits obtained through the use of parallelism [Gerb87, Cope88]. In this case, disks can be partitioned into relation-clusters and the data placement algorithm described above can be modified to include a term, Disk_Start(R), which indicates the starting disk of the relation-cluster occupied by relation R. In addition, the total number of disks, M, in the original formulas is replaced by SRC, the size of the **relation-cluster**. The modified data placement algorithm will be as follows. The i-th primary fragment is placed on the

{ Disk_Start(R) + [C(R) + i] mod SRC }-th disk,

and the i-th backup fragment is placed on the

{ Disk_Start(R) + [C(R) + i + 1] mod SRC }-th disk.

The disks in a **relation-cluster** can be further divided into even smaller groups and each of them will form a **chain-cluster** which is the unit of load balancing in the event of a node failure. Under this grouping mechanism, the formula for placing the i-th primary and backup fragments respectively will be further generalized to:

{ CC_Off(k) + Disk_Start(R) + [C(R) + i] mod SCC }, and

{ CC_Off(k) + Disk_Start(R) + [C(R) + i + 1] mod SCC }.

SCC is the size of **chain-cluster** and CC_Off(k) is the offset (in terms of the number of disks) of the k-th **chain-cluster** within the enclosing **relation-cluster**.

The generalized chained declustering algorithm declusters a relation over any SRC disks out of a total of M disks (2 <= SRC <= M), and "chains" a group of SCC disks(2 <= SCC <= SRC) by the identical primary and backup fragments. For the extreme case of SCC = 2, the resulting distribution of tuples is identical to the mirrored disk scheme. When 2 < SCC <= M, chained declustering is similar to, but, as will be demonstrated later, provides higher data availability than the interleaved declustering scheme. Figure 3.2 illustrates data placement for the generalized chained declustering scheme with SCC = 4, SRC = 8, and M = 16. In Figure 3.2, R_i (S_i) and r_i (s_i) respectively represent the i-th primary fragment and the i-th backup fragment of relation R (S).

	Relation-cluster 0								**Relation-cluster 1**							
	Chain-cluster 0				**Chain-cluster 1**				**Chain-cluster 2**				**Chain-cluster 3**			
Node	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Primary Copy	R0	R1	R2	R3	R4	R5	R6	R7	S0	S1	S2	S3	S4	S5	S6	S7
Backup Copy	r3	r0	r1	r2	r7	r4	r5	r6	s3	s0	s1	s2	s7	s4	s5	s6

Figure 3.2: Chained Declustering (General Case)

In the following sections, for purposes of simplicity, we assume that a relation-cluster contains all of the disks in the system and that it is not subdivided into multiple chain-clusters.

3.4. Failure Handling with Chained Declustering

In the case of a single node (processor or disk) failure the chained declustering strategy is able to uniformly distribute the workload of the cluster among the remaining operational nodes. For example, with a cluster size of 8, when a processor or disk fails, the load on each remaining node will increase by 1/7th. Figure 3.3 illustrates how the workload is balanced in the event of a node failure (node 1 in this example) with the chained declustering mechanism. During the normal mode of operation, read requests are directed to the fragments of the primary copy and write operations update both copies. When a failure occurs, pieces of both the primary and backup fragments are used for read operations. For example, with the failure of node 1, primary fragment R1 can no longer be accessed and thus its backup fragment r1 on node 2 must be used for processing queries that would normally have been directed to R1. However, instead of requiring node 2 to process all accesses to both R2 and r1, chained declustering offloads 6/7ths of the accesses to R2 by redirecting them to r2 at node 3. In turn, 5/7ths of access to R3 at node 3 are sent to r3 instead. This dynamic reassignment of the workload results in an increase of 1/7th in the workload of each remaining node in the cluster. Since the relation-cluster size can be increased without penalty, it is possible to make this load increase as small as is desired.

Node	0	1	2	3	4	5	6	7
Primary Copy	R0	---	$\frac{1}{7}$R2	$\frac{2}{7}$R3	$\frac{3}{7}$R4	$\frac{4}{7}$R5	$\frac{5}{7}$R6	$\frac{6}{7}$R7
Backup Copy	$\frac{1}{7}$r7	---	r1	$\frac{6}{7}$r2	$\frac{5}{7}$r3	$\frac{4}{7}$r4	$\frac{3}{7}$r5	$\frac{2}{7}$r6

Figure 3.3: Fragment Utilization with Chained Declustering
After the Failure of Node 1 (Relation-Cluster Size = 8)

What makes this scheme even more attractive is that the reassignment of active fragments incurs neither disk I/O nor data movement. Only some of the bound values and pointers/indices in a memory resident control table must be changed, and these modifications can be done very quickly and efficiently.

3.5. Load Balancing Algorithm

The example shown in Figure 3.3 provides a simplified view of how the chained declustering mechanism actually balances the workload in the event of a node failure. In actual database applications, however, queries cannot simply access an arbitrary fraction of a data fragment because data may be clustered on certain attribute values, indices may exist, and the query optimizer may generate different access plans. For example, Gamma [DeWi86] provides the user with three declustering alternatives: range, hash, and round-robin partitioning. For each of these partitioning alternatives, Gamma also supports five storage organizations for each relation[2]:

(1) a clustered index on the partitioning attribute,

(2) a clustered index on a non-partitioning attribute,

(3) a non-clustered index on the partitioning attribute,

(4) a non-clustered index on a non-partitioning attribute,

(5) heap (no index).

In addition, three alternative access plans are generated by Gamma's query optimizer for processing the tuples of one relation. These are: (1) utilize the clustered index, (2) utilize a non-clustered index, and (3) perform a sequential file scan.

In this section, we describe the design of a load balancing algorithm for the chained declustering mechanism that can handle all possible combinations of partitioning methods, storage organizations, and access plans. The keys to the solution are the notion of a **responsible range** for indexed attributes, the use of **query modification techniques** [Ston75], and the availability of an **extent map** for relations stored as a heap. In the following sections, we will describe these techniques in more detail and illustrate how they are used.

3.5.1. Responsible Range and Extent Map

For each indexed attribute, W, we term the range of values that a node is responsible for its **responsible range on attribute W** (or its responsible W range). The responsible range for each fragment is represented by the interval of two attribute values and is stored in a table termed the **active fragment table** which is maintained by the

[2] All fragments of a relation must have the same "local" storage organization.

query scheduler[3]. The following formulas are used to compute the responsible range for each relation with one or more indexed attributes. When node S fails, node j's responsible range on attribute W for a primary fragment R_i is:

$$\left[W_l(R_{i)},\ \ W_h(R_{i)} - f(j,S) * (W_h(R_{i)} - W_l(R_{i)+1})\right],\quad \text{where } f(j,S) = \frac{[M-(j-S+1)] \bmod M}{M-1},$$

and the responsible range on attribute W for a backup fragment r_i is:

$$\left[W_l(r_{i)} + g(j,S) * (W_h(r_{i)} - W_l(r_{i)+1}),\ \ W_h(r_{i)}\right],\quad \text{where } g(j,S) = \frac{[M-(S-j+1)] \bmod M}{M-1}.$$

M is the total number of nodes in the relation-cluster. $W_l(R_{i)}$ and $W_h(R_{i)}$ correspond to the lower and upper bound values of attribute W for fragment R_i. For example, if M=4 and S=1, then f(2,S)= 2/3. Assuming that $W_l(R2)$ is 201 and $W_h(R2)$ is 300, when node 1 fails the responsible range of node 2 on attribute W for its primary fragment R2 is

$$\left[201,\ \ 300 - 2/3 * 100\right] = \left[201,\ \ 233\right].$$

When a relation is stored as a heap, each fragment is logically divided into M-1 extents and the physical disk address of the first and last page in each extent is stored in a table termed the **extent map**. The responsible ranges (extents) of fragments without any indices are marked by two extent numbers (e.g. extents 1 to 3). The responsible range of node j for a fragment is determined by the following formulas:

For the primary fragments, extents $\left[E_1\ ,\ \ E_{(M-1)*(1-f(j,S))}\right]$

For the backup fragments, extents $\left[E_{1+(M-1)*g(j,S)}\ ,\ \ E_{M-1}\right]$.

Here, E_i is the i-th extent of a fragment and functions f and g are as previously defined.

Figure 3.4 illustrates the structure of an extent map. As shown in Figure 3.4, the extent map for a fragment (R_i) of any relation R at any node A is an array of M-1 records. Each record has two fields which contain the addresses of the first and last pages of an extent. When the sectors of a fragment are uniformly distributed over M-1 extents, the i-th extent will consist of sectors $\left[\frac{(i-1)*T}{(M-1)} + 1,\ \ \frac{i*T}{(M-1)}\right]$, where T is the total number of sectors occupied by fragment R_i at node A. Figure 3.5 illustrates the responsible extents of each node before and after a node failure for a relation that is stored as a heap.

[3] The query scheduler in Gamma is also responsible for scheduling multisite queries and activating operator processes used to execute the nodes of a compiled query tree.

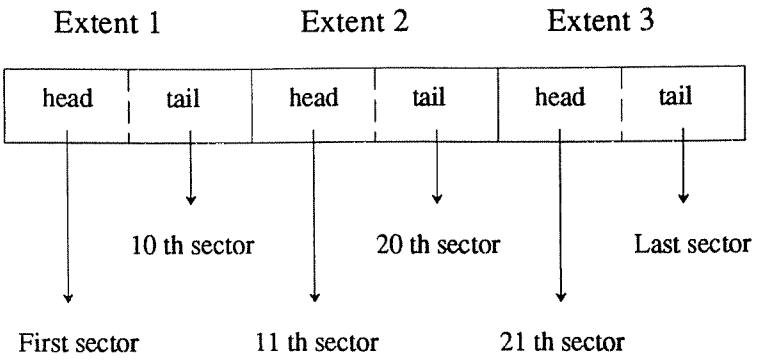

Figure 3.4: An example of an extent map with M=4 and T=30.

Node	0	1	2	3
Primary Copy	R0	R1	R2	R3
Extents:	1-3	1-3	1-3	1-3
Backup Copy	r3	r0	r1	r2
Extents:	1-3	1-3	1-3	1-3

a) Normal Mode of Operation

Node	0	1	2	3
Primary Copy	R0	---	R2	R3
Responsible Extents:	1-3	---	1	1-2
Backup Copy	r3	---	r1	r2
Responsible Extents:	3	---	1-3	2-3

b) Operation after the failure of node 1

Figure 3.5: Relation Stored as a Heap.

In the normal mode of operation, primary fragments are designated as the active fragments and the responsible ranges are set accordingly - either to the range of an indexed attribute or to extents 1 to M-1 (M is the number of nodes in the relation-cluster). When a node failure occurs, the responsible ranges and, thus, the active fragments are reassigned. In this event, the active fragments will consist of both primary **and** backup fragments at each of the operational nodes. The reassignment of the responsible ranges in the event of a node failure is done in such a way that the sizes of the active fragments for each of the remaining nodes within the relation-cluster will be increased by 1/(M-1)th. This, in turn, will increase the workload of all active nodes within the relation-cluster by 1/(M-1)th when queries exhibit a uniform access pattern to the nodes. If queries exhibit a non-uniform access pattern, the functions f and g defined above can easily be modified to capture the skewed data access pattern.

This section illustrated how the responsible ranges and active fragments are determined and marked. In the next section, we will show how this information is used by the query scheduler to modify and process queries in the event of a node failure.

3.5.2. Query Modification and Processing

In order to ensure that only the active fragments are accessed when a failure occurs, some queries must be modified before being sent to local nodes for processing. For indexed relations, two query modification techniques are used. The first technique modifies the range of a selection predicate to reflect the responsible range of a node before sending the query to the node. This technique is used when an index is used to retrieve qualifying tuples. The second query modification technique adds a selection predicate to a query based on the responsible range of the attribute on which the clustered index exists. This technique is used when a sequential file scan is selected by the query optimizer.

Query modification techniques, however, can only be applied to indexed relations. For relations stored as a heap, a different technique is needed to ensure that only the appropriate fragments of the primary and backup copies are accessed. In this situation, the extent map described in Section 3.5.1 will be used.

Algorithm A: Changing the Selection Predicate

For selection queries in which an index on the qualified attribute W is used to retrieve qualifying tuples, the load balancing algorithm works as follows:

> The query scheduler first computes the **intersection** of the range of the selection predicate on attribute W and the responsible range of node A on attribute W. If the **intersection** of the two ranges is non-empty, the query will be modified to have the **intersection** as its new range, and the modified query is sent to the node A. An exact match selection query is treated as a special case of range queries where the range of the selection predicate is a single point.

Algorithm B: Adding a Selection Predicate

When a suitable index is not available, the query must be processed by sequentially scanning all active fragments. Since no data is moved when the active fragments are reassigned after a node fails, the only way to distinguish the active and passive fragments of the primary and backup copies at a node is through the range of the attribute with the clustered index. Assuming that a clustered index exists on attribute X, the load balancing algorithm works as follows:

> For each active fragment i of a node A, fetch its responsible range for attribute X (e.g. $X_1(R_i)$ to

$X_h(R_i)$). A selection predicate on attribute X corresponding to the current responsible X range of fragment i (i.e. $X_l(R_i) \leq R.X \leq X_h(R_i)$) is appended to the original query before the query is sent to node A. After receiving a query, node A retrieves tuples sequentially from $X_l(R_i)$ to $X_h(R_i)$ because the tuples in its fragment are sorted on their X attribute values.

Algorithm C: Using the Extent Map

When a relation is stored as a heap, the only way to retrieve tuples is via a sequential file scan. The active fragment table and the extent map described in Section 3.5.1 will be used for determining the responsible ranges (extents) and, thus, the active fragments in the event of a node failure. Under this condition, queries will be processed in the following manner:

With this type of query, the addresses of the first and last pages in an extent (which are stored in the extent map) are used by a node to perform a sequential scan on the extent. Upon receiving a query from the query scheduler, a node will sequentially retrieve tuples from the appropriate extents. In the normal mode of operation, a node will access only the extents of the primary fragment. In the case of a node failure, the active fragments will consist of fragments from both the primary and backup copies. The retrieval process for a node is completed when all tuples within its responsible range have been accessed.

3.5.3. Applications of the Load Balancing Algorithms

To illustrate the operation of the load balancing algorithms, assume that relation R, with attributes X, Y and Z, is horizontally partitioned using the chained declustering strategy, that X is always used as the partitioning attribute, and that attributes X and Z have values ranging from 1 to 400 and 1 to 300, respectively. We further assume that there are four nodes in the system, that the relation-cluster consists of four nodes (M=4), and that the second node (S=1) has failed.

Example #1: Clustered index on range partitioning attribute is used to retrieve the qualified tuples.

Assume that relation R is range partitioned on attribute X and that a clustered index has also been constructed on X. As shown in Figure 3.6a, the primary and backup fragments of relation R have disjoint sets of X attribute values. For exact match selection queries on attribute X (e.g. R.X = constant), the access plan will use the clustered

index on R and at run time the query scheduler will direct the query to the appropriate node for execution. For example, given the range of X attribute values shown in Figure 3.6, queries with predicates R.X = 150, R.X = 240, and R.X = 395 will be directed, respectively, to nodes 1, 2, and 3 during the normal mode of operation and to nodes 2, 3, and 0 in the event that node 1 has failed.

Node	0	1	2	3
Primary Copy X Values:	R0 1-100	R1 101-200	R2 201-300	R3 301-400
Backup Copy X Values:	r3 (301-400)	r0 (1-100)	r1 (101-200)	r2 (201-300)

a) Normal Mode of Operation

Node	0	1	2	3
Primary Copy Responsible X Values:	R0 1-100	--- ---	R2 201-233	R3 301-366
Backup Copy Responsible X Values:	r3 367-400	--- ---	r1 101-200	r2 234-300

b) Operation after the failure of node 1

Figure 3.6

Handling queries with range predicates is slightly more complicated. In this case, the query scheduler modifies the range of the predicate in the compiled query predicate appropriately (i.e. Algorithm A in Section 3.5.2 is employed). For example, the query

retrieve ... where R.X > 150 and R.X < 250

would normally be sent by the query scheduler to nodes 1 and 2 for processing. If the same query is submitted while 1 is unavailable, the scheduler will modify the compiled query plan to produce the following two queries:

Q1: retrieve ... where R.X ≥ 201 and R.X ≤ 233 or r.X > 150 and r.X ≤ 200
Q2: retrieve ... where r.X ≥ 234 and r.X < 250

Q1 will be sent to node 2 for processing and Q2 will be directed to node 3.

Example #2: Clustered index on hash partitioning attribute is used to retrieve the qualified tuples

In this example, we assume that relation R is hash partitioned on attribute X and that a clustered index has also been constructed on X. With the hashed partitioning strategy, as each tuple is loaded into the database, a hash

function is applied to the partitioning attribute and the resulting value is used to select a storage unit. Currently, the hash function used in Gamma generates 32-bit hash values when it is applied to the partitioning attribute, X, of a tuple. The hash value, h(X), is then divided by the number of storage units, M, and the remainder, rem(X), is used to select a storage unit to store the corresponding tuple. The quotient, q(X), which has values ranging from 0 to q_{max} ($q_{max} = (2^{32}-1)/M$) is used to determine the responsible h(X) range for each fragment. In this case, the responsible range of each active node consists of two values: the hash value of attribute X and the actual value of attribute X. The hash value of attribute X is used by query scheduler to select the responsible node for exact match selection queries while the actual X value is used when processing range queries and file scans.

In the normal mode of operation, the primary fragments are designated as the active fragments and the responsible X range is set accordingly. The responsible ranges on h(X) are set to "0 to q_{max}" for all primary fragments. When a failure occurs, the responsible X and h(X) ranges are recomputed for each fragment using the formulas described in Section 3.5.1 (when recomputing the responsible range on h(X), 0 is substituted for W_l and q_{max} is substituted for W_h). Figure 3.7 illustrates the assignment of responsible ranges before and after a node failure. As before, we assume that there are 4 (M=4) nodes, that the second node has failed, and that attribute X has values ranging from 1 to 400. For purposes of this example, assume that the hash function h generates only a 16-bit hash value (thus, q_{max} = 16383). For simplicity we use "1 to 400" as the X range for all fragments. In practice, different fragments will have slightly different X ranges and each of them will be a subrange of "1 to 400".

For exact match selection queries on attribute X the query will be directed to a single node where it will be processed using the clustered index. The query scheduler first applies the hash function to the constant X in the selection predicate and the outcome, h(X), is divided by M to generate quotient q(X) and remainder rem(X). rem(X) indicates where the qualified tuple(s) is stored and q(X) is used to select either the primary or the backup fragment according to the responsible h(X) range. For example, if h(X=97) is equal to 30770, then q(X=97) will be equal to 30770/4 = 7692 and rem(X=97) will be equal to (30770 **mod** 4) = 2. rem(X)= 2 indicates that the qualified tuple(s) is in the second fragment (tuples hashed to rem(X)= i are stored in the i-th fragment). Because the second fragment is stored at two nodes (the primary copy is stored at node 2 and the backup copy is stored at node 3), q(X) is used to select a node by comparing the responsible h(X) ranges of these two nodes for the second fragment. According to Figure 3.7b, q(X) is within node 3's responsible h(X) range for the second fragment ("5461 to q_{max}" for r3), so the query "R.X = 97" will be modified to "r.X = 97" before it is sent to node 3 for processing. Upon

Node	0	1	2	3
Primary Copy	R0	R1	R2	R3
X Values:	1-400	1-400	1-400	1-400
h(X) Values:	0-q_m	0-q_m	0-q_m	0-q_m
Backup Copy	r3	r0	r1	r2
X Values:	(1-400)	(1-400)	(1-400)	(1-400)
h(X) Values:	(0-q_m)	(0-q_m)	(0-q_m)	(0-q_m)

a) Normal Mode of Operation ($q_m = (2^{16}-1)/M = 16383$)

Node	0	1	2	3
Primary Copy	R0	---	R2	R3
Responsible X Values:	1-400	---	1-133	1-266
Responsible h(X) Values:	0-q_m	---	0-5460	0-10921
Backup Copy	r3	---	r1	r2
Responsible X Values:	267-400	---	1-400	134-400
Responsible h(X) Values:	10922-q_m	---	0-q_m	5461-q_m

b) Operation after the failure of node 1

Figure 3.7: Assignment of Responsible Ranges for Hash Partitioning Attribute X.

receiving a query, a local node will use the clustered index to retrieve qualified tuple(s).

Example #3: Index on non-partitioning attribute is used to retrieve the qualified tuples.

Assume that a non-clustered index is constructed on attribute Z and that attribute Z has values ranging from 1 to 300. Figure 3.8 depicts the responsible range of the nodes before and after node 1 has failed. Because each of the fragments have the same range of Z values, after the failure of node 1 and the reassignment of responsible ranges, the responsible range of each of the remaining active nodes will still be equal to the full range of Z attribute values. Thus, the query must be sent to all active nodes for processing. At each node the responsible ranges for both the primary fragment and the backup fragment will vary, however, depending on exactly which node has failed.

For range selection queries on attribute Z for which the access plan uses the non-clustered index, the query scheduler will use Algorithm A from Section 3.5.2 to modify the query before it is sent to the appropriate nodes for processing. Figure 3.9 illustrates how a range query is modified after the failure of node 1. The query is sent unmodified to node 0 because the query range is a subrange of node 1's responsible Z range for the primary fragment while the intersection of the query range and node 0's responsible Z range for the backup fragment is empty. For node 3, the query range is again a subrange of its responsible Z range for the primary fragment, but the intersection of the query range and its responsible Z range for the backup fragment is non-empty (from 101 to 110). As a

Node	0	1	2	3
Primary Copy	R0	R1	R2	R3
Z Values:	1-300	1-300	1-300	1-300
Backup Copy	r3	r0	r1	r2
Z Values:	(1-300)	(1-300)	(1-300)	(1-300)

a) Normal Mode of Operation

Node	0	1	2	3
Primary Copy	R0	---	R2	R3
Responsible Z Values:	1-300	---	1-100	1-200
Backup Copy	r3	---	r1	r2
Responsible Z Values:	201-300	---	1-300	101-300

b) Operation after the failure of node 1.

Figure 3.8: active fragment table for attribute Z

result, the query is modified to include both intersections before it is sent to node 3. The query modification for node 2 is done in a similar fashion.

retrieve ....
where R.Z ≥ 90 **and** R.Z ≤ 110

a) Original query

retrieve ....
where R.Z ≥ 90 **and** R.Z ≤ 100
or r.Z ≥ 90 **and** r.Z ≤ 110

c) Query sent to node 2

retrieve ....
where R.Z ≥ 90 **and** R.Z ≤ 110

b) Query sent to node 0

retrieve ....
where R.Z ≥ 90 **and** R.Z ≤ 110
or r.Z ≥ 101 **and** r.Z ≤ 110

d) Query sent to node 3

Figure 3.9: Query modification on non-partitioning attribute

3.5.4. Summary

The idea behind the algorithm described in this section is to process queries against a set of active fragments which are initially designated when the relation is created and are then reassigned when node failures occur. For a given relation, different queries may use different active fragments, but the union of all active fragments used to process a query is guaranteed to correspond to a complete copy of the relation. For each query, the query scheduler selects the responsible node(s) based on the current assignment of active fragments, and each node retrieves only those tuples within its responsible range.

Table 3.1 summarizes the operation of the load balancing algorithm for all possible combinations of partitioning methods, storage organizations, and access plans. In Table 3.1, X is the partitioning attribute. The first and second columns correspond to the cases where the clustered index is on the partitioning attribute X and a non-partitioning attribute, respectively, whereas the third column describes how queries are processed when the relation is stored as heap. A, B, and C stand for three load balancing algorithms described in Section 3.5.2, whereas e, f, g, and h are used to indicate the set of nodes that may be responsible for processing a given query. For example, if relation R is range partitioned on attribute X and the clustered index is also on attribute X, for a query plan that accesses data via a non-clustered index (first column, second row), algorithm A will be used to modify the query plan in the failure mode of operation. The "e" next to "A" indicates that the modified query plan will be sent to all active nodes for processing. This is because the qualified attribute (the one with a non-clustered index) is not the partitioning attribute X (which has the clustered index). On the other hand, if the clustered index is on a non-partitioning attribute (second column of second row), the same load balancing algorithm, A, will be applied to modify the query plan in the failure mode. However, the modified query plan will be sent to all active nodes (indicated by "e") if the qualified attribute is not the range partitioning attribute X, or it may only be sent to a group of selected nodes (indicated by "g") if the qualified attribute is also the range partitioning attribute X.

RANGE PARTITIONING	Clustered index on the partitioning attribute	Clustered index on a non-partitioning attribute	No index
uses the clustered index	A, f	A, e	N/A
uses a non-clustered index	A, e	A, e/g	N/A
uses a file scan	B, e	B, e/g	C, e/g
HASH PARTITIONING			
uses the clustered index	A, e/h	A, e	N/A
uses a non-clustered index	A, e	A, e/h	N/A
uses a file scan	B, e	B, e/h	C, e/h
ROUND ROBIN			
uses the clustered index	A, e	A, e	N/A
uses a non-clustered index	A, e	A, e	N/A
uses a file scan	B, e	B, e	C, e

Table 3.1: Summary of Load Balancing Algorithm

A: modify query range based on the **intersection** of the responsible range and the original selection predicate of the query
B: append a range selection predicate on the attribute with the clustered index
C: use extent map
e: send query to all active nodes within the relation-cluster
f: send query to a subset of the active nodes
g: select nodes based on the value of X if X is the qualified attribute
h: select node based on responsible h(X) range for exact match queries
N/A: not applicable

CHAPTER 4

AVAILABILITY VS. LOAD BALANCING - AN ANALYSIS

4.1. Introduction

In a distributed computing environment, balancing the load among all nodes is regarded as one of the most important issues to be solved [Eckh78]. For computation intensive (CPU bound) applications, one common method for balancing the load among all processors is process migration, i.e., moving processes from highly loaded nodes to lightly loaded nodes [Livn82, Eage86]. For data intensive (I/O bound) applications, simple process migration is not generally a good solution because processes are most efficiently run on those nodes containing the data referenced by the query. For these latter (data intensive) applications, a different strategy is needed to balance the workload among all nodes. Query processing on multi-processor database machines, like Gamma [DeWi86, DeWi90], the Teradata DBC/1012 [Tera85], Tandem's parallel NonStop SQL [Engl89], and Bubba [Cope88] is an example of such a data intensive application. In these systems, load balancing is achieved through the use of data declustering and intra query parallelism.

With replicated data, declustering and parallelism alone do not, however, guarantee load balancing in the event of node failures. Some other action must be taken in order to balance the load among the remaining operational nodes. However, not all data replication schemes offer the same degree of load balancing when operating in the presence of failures. Worse yet, while availability can be improved through the use of data replication and performance can be improved through the use of declustering and parallelism, as illustrated in [Tera85], it is not a straightforward task to simultaneously achieve both objectives in the event of processor or disk failures. In the following sections we will examine, using an analytical model, the degree of data availability and the level of load balancing provided by different data replication schemes.

4.2. Availability

For the purpose of the following availability and performance analyses, we have made the following assumptions:

- A relation is partitioned across M disks,
- A relation-cluster consists of only one chain-cluster and its size is also M,
- The cluster size of the interleaved declustering scheme is N,
- The group size of RAID is G.

Let n_1 be the number of ways that the first disk (out of M disks) can fail and let n_2 be the number of ways that a second disk failure, together with the first failure, will result in data being unavailable. Then n_1 is the number of possible combinations of selecting one out of M disks, i.e. $n_1=\binom{M}{1}$ = M. n_2, however, depends on the data placement (declustering) scheme used.

With Teradata's interleaved declustering scheme, if the second failure is in the same cluster as the first one, data will become unavailable. Because there are N-1 disks remaining in the failed cluster after the first failure, n_2 is then $\binom{N-1}{1}$ = N-1. The case for RAID is similar. If the second failure is in the same RAID group as the first failed disk, some data will become unavailable. So $n_2 = \binom{G-1}{1}$ = G-1. For mirrored disks, after the first failure, data will be unavailable only if the mirror image of the failed disk also fails. Because every disk has exactly one mirror image, n_2 will be $\binom{1}{1}$ = 1. With chained declustering, data will be unavailable only if two logically adjacent disks fail, and since all disks have two neighbors, the number of different ways for this to happen, i.e. n_2, is equal to $\binom{2}{1}$ = 2.[1]

Let E be the total number of events for which the failure of two disks will cause data to become unavailable, then for a system with M disks, E(Scheme)= $n_1 * n_2$. For the different schemes considered in this dissertation, their respective E values are:

$$\text{E(chained)} = \binom{M}{1} * 2 = 2M,$$

$$\text{E(mirrored)} = \binom{M}{1} * 1 = M,$$

$$\text{E(interleaved)} = \binom{M}{1} * \binom{N-1}{1} = M*(N-1),$$

$$\text{E(RAID)} = \binom{M}{1} * \binom{G-1}{1} = M*(G-1).$$

[1] This is true except when the size of a chain-cluster is equal to 2. In that case, every disk has only one "logical" neighbor and n_2 will be equal to 1.

Let PDU be the probability of any data being unavailable. Then PDU will be equal to E multiplied by p, the probability of two disks failing at the same time (i.e. PDU(scheme) = E(scheme) * p). The PDU values for the schemes considered in this dissertation will then be:

PDU(chained) = 2M*p,

PDU(mirrored) = M*p,

PDU(interleaved)= M*(N-1)*p,

PDU(RAID) = M*(G-1)*p.

As one can see from the analysis above, the mirrored disk scheme provides the highest degree of data availability[2] while the chained declustering scheme is second. The PDU of both of these schemes is a linear function of the total number of disks, M. In contrast, as indicated in the formula above, the PDU value of Teradata's interleaved declustering scheme varies significantly depending on the cluster size, N. At one end, when N is 2, interleaved declustering has the same degree of data availability as mirrored disks. At the other end of the spectrum, when N is approaching M, the PDU of interleaved declustering is proportional (approximately) to M squared. When N=8, the probability of data being unavailable with interleaved declustering will be 350% higher than with the chained declustering scheme. When N=32, it will be 1550% higher. RAID has the same availability as the interleaved declustering scheme if the group size, G, is equal to the cluster size, N. While Teradata suggests that its customers should select a cluster size of 4 or 8 in order to achieve a reasonable balance between performance and availability, RAID assumes that G may be 10 or 25 in its design analysis [Patt88]. With G = 25, RAID will be 24 times more likely to have data unavailable than mirrored disks and 12 times more likely than chained declustering.

4.3. Load Balancing and Performance

Although the mirrored disk scheme provides the highest degree of data availability, its major drawback is that it is not able to balance the workload when a node (processor or disk) fails. For example, assume that file A is stored on two mirrored disk pairs attached to processors 1 and 2. When either processor 1 or 2 fails, the remaining processor will be responsible for all accesses to both mirrored pairs and the workload on the remaining processor will double. In addition, when a disk in a mirrored pair fails, the remaining disk in the failed pair will be

[2] Except, of course, when N, G, and the size of the chain-cluster are all equal to 2. In that case, all three of these schemes degenerate to the mirrored disk scheme and the PDU of all schemes will be identical.

responsible for all requests originally served by the two disks. The increase in workload on the remaining disk is proportional to the ratio of disk read/write requests. The higher the read ratio is, the higher the load increase on the remaining disk will be. In the worst case, the workload of the remaining disk will double.

In contrast, in the case of a single node (processor or disk) failure both the chained and interleaved declustering strategies are able to uniformly distribute the workload of the failed cluster among the remaining operational nodes. For example, with a cluster size of 7, when a processor or disk fails, the load on each remaining node will increase by 1/7. One might conclude then that the cluster size should be made as large as possible (until, of course, the overhead of parallelism starts to overshadow the benefits obtained). While this is true for chained declustering, the availability of the interleaved declustering strategy is inversely proportional to the cluster size (as described in Section 4.2). Thus, for example, doubling the cluster size in order to halve (approximately) the increase in the load on the remaining nodes when a failure occurs has the negative side effect of doubling the probability that data will actually be unavailable. This is the reason that Teradata recommends limiting the cluster size to 4 or 8 processors.

In the RAID data placement scheme, the system will be restricted to one disk I/O per recovery group (G disks) when attempts are made to access data on the failed disk before it is repaired/replaced. This is true because all G-1 remaining disks in the same group must be read in order to regenerate the bad sector. In addition, to reconstruct a failed disk, all sectors (both check and data sectors) on the other disks in the same group must be read and manipulated ("exclusive-or"-ed, byte by byte). This reconstruction process will compete with normal processes for both CPU and I/O bandwidth. System performance, in terms of both throughput and response time, will therefore degrade significantly during the reconstruction period. With 1000 disks in a system, a disk will fail about every 30 hours. Therefore, when 1000 disks are employed in a RAID system, some group will undergo reconstruction almost every day.

4.4. Summary

In Table 4.1 we summarize the probability of data being unavailable and the impact of a node failure on the workload of the remaining nodes. As indicated in the table, both chained declustering and interleaved declustering are able to evenly balance the workload when a node fails. Nevertheless, to make the interleaved declustering scheme achieve the same level of load balancing in the event of a node failure as the chained declustering scheme, its cluster size, N, must be set to M (i.e., 32). With N = 32, however, the interleaved declustering scheme will be

about 16 times more likely to have data unavailable. On the other hand, to make the interleaved declustering scheme provide the same degree of data availability as the chained declustering scheme, its cluster size, N, must be set to 3. In that case, however, the load increase at the remaining nodes in the failed cluster will be about 16 times higher than that of the chained declustering scheme in the event of a node failure.

With current disk technology, the mean time to failure for a disk drive is between 3 and 5 years. If we assume that failures are independent, that the time between failures is exponentially distributed with a mean time to failure of 3 years, and that the time to repair/replace a disk and restore its contents is 5 hours, then for a pair of disks (M =

Schemes	**Chained declustering**	**Mirrored disks**	**RAID**	**Interleaved declustering**
Probability of any data being unavailable	2M**p*	M**p*	(G-1)M**p*	(N-1)M**p*
Load increase on remaining nodes	$\frac{1}{M-1}$	100%	(1)	$\frac{1}{N-1}$
M=N=G=32				
Probability of any data being unavailable	64**p*	32**p*	992**p*	992**p*
Load increase on remaining nodes	3.23%	100%	(1)	3.23%
N = G = 8				
Probability of any data being unavailable	64**p*	32**p*	224**p*	224**p*
Load increase on remaining nodes	3.23%	100%	(1)	14.3%
N = G = 4				
Probability of any data being unavailable	64**p*	32**p*	96**p*	96**p*
Load increase on remaining nodes	3.23%	100%	(1)	33.3%
N = G = 3				
Probability of any data being unavailable	64**p*	32**p*	64**p*	64**p*
Load increase on remaining nodes	3.23%	100%	(1)	50.0%

(1) Accesses to sectors on the failed disk will be quite expensive, as all remaining disks in the "failed" group must be accessed in order to regenerate the requested data.

Table 4.1: Summary of Availability and Load Balancing (M=32)

2), the probability, p, of both disks failing at the same time is about 0.0002.[3] For M = 8, M = 32, M = 250, and M = 1000, the probability of any two disks failing at the same time will be 0.0014M, 0.0062M, 0.046M, and 0.17M, respectively. If the failure of any two disks at the same time causes data to be unavailable, some data will be unavailable once every 15.1 years[4] when M = 32, once every 3.2 months when M = 250, and once every 6.4 days when M = 1000. With chained declustering, as described in Section 3.2, a relation will be unavailable only if two (logically) adjacent disks within a chain-cluster fail simultaneously, or if the second disk fails while the first one is being replaced. The probability of any data being unavailable is thus 0.0004M for all M > 2. As a result, for a relation declustered over 1000 disks using the chained declustering strategy, some data will be unavailable once every 7.5 years instead of every 6.4 days.

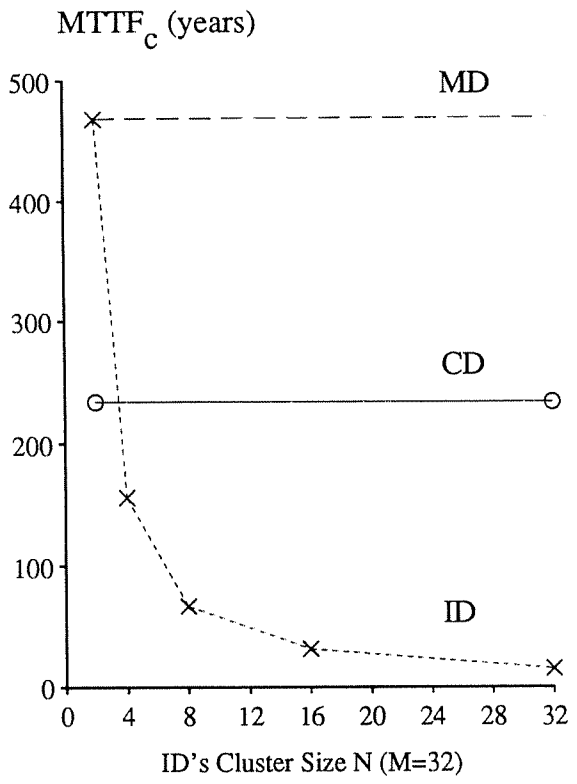

Figure 4.1: MTTF$_c$ vs. Cluster Size

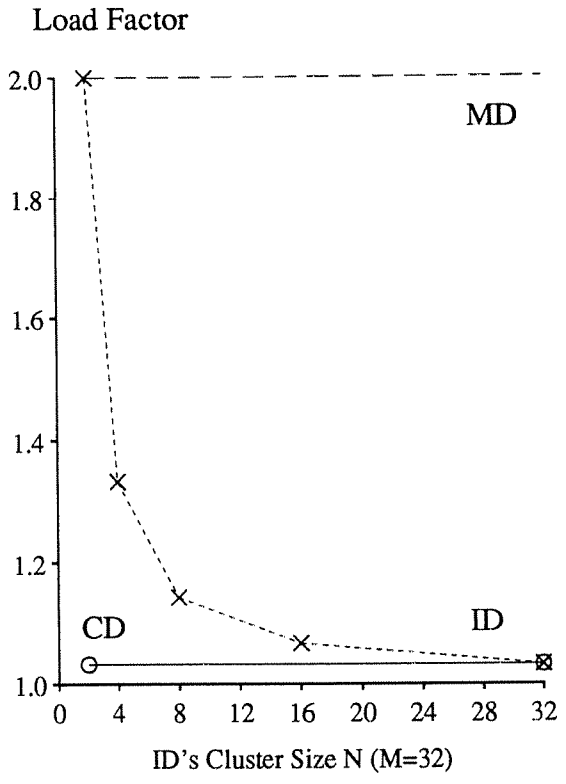

Figure 4.2: Load Increase vs. Cluster Size

[3] See the Appendix for the derivation of this and other p values under different assumptions of MTTF (disk) and disk repair time.

[4] This considers disk failures only. If failures of all other components which may cause unavailability are included, the probability of unavailable data will be higher.

With p = 0.0002 and M = 32, Figures 4.1 and 4.2 illustrate, respectively, the mean time between cluster failure ($MTTF_c$)[5] and the excess load on the remaining nodes when one node fails. The interleaved declustering curve in Figure 4.1 illustrates that, for a system containing 32 nodes, the mean time between cluster failures ($MTTF_c$) decreases dramatically as the cluster size is increased from 2 to 32 nodes. In other words, an increase in the cluster size reduces the probability that all data will be available. On the other hand, the interleaved declustering curve in Figure 4.2 demonstrates that larger cluster sizes have the beneficial effect of reducing the increase in the workload on the remaining nodes when a node in a cluster fails. Figures 4.1 and 4.2 also show that the mirrored disk scheme provides very good data availability, but that it does poorly in balancing the workload during an outage. In contrast, the chained declustering scheme provides both reasonably good data availability and excellent load balancing.

[5] The $MTTF_c$ is defined as two or more disks in a cluster being down at the same time.

CHAPTER 5

SIMULATION MODEL DESCRIPTION

5.1. Introduction

Comparing the performance of different computer systems through direct measurement is perhaps the most accurate method of comparison. One problem with this method, however, is that it is possible only when the systems to be measured are already operational. And, even if the systems are already operational, there are other obstacles in the way of doing direct measurement. The major obstacle is that direct measurement is usually a computationally expensive procedure. It may take a long time (both CPU time and real time) to obtain the desired results. A less severe obstacle is that direct measurement has to be carried out in a controlled manner. This usually means that such measurement needs to be done either late at night or on a holiday when no other users are on the system[1]. An alternative for evaluating the performance of computer systems is through analytical modeling. This method is cost-effective, but accuracy is almost always sacrificed for simplicity of the equations in order to make the solutions tractable.

Our goal is to investigate the relative performance of different high availability strategies in a multi-user database environment. Neither of the above two approaches seems appropriate because we do not have enough hardware and software resources for direct measurement, and the problem appears too complex for a detailed analytical model. Thus, simulation was chosen for evaluating the performance of three selected high availability strategies in this study.

Our simulation model is based on the hardware and software architectures of the Gamma database machine [DeWi90] running on a 32 node Intel iPSC/2 hypercube. In our model, a Database Machine is constructed from 12 primitive components: a **Database Manager**, which first initializes the database and then stores file and page mapping information in a system catalog; a **Terminal**, which generates the workload of the system; a **Query**

[1] Unless, of course, you alone own the full system and its continued availability is not critical.

Manager, which compiles queries and constructs query plan trees; a **Scheduler**, which activates operator processes at the Operator Manager module and coordinates the execution of all queries; a **Network Manager**, which models a simplex, fully-connected, point-to-point communication network; a **Network Interface**, which models the network interface of a query processing node; a **CPU Manager**, which models the sharing of the CPU resource of a node; an **Operator Manager**, which implements the details of query (or sub-query) execution at a node; a **Disk Manager**, which models the behavior of a disk I/O process; a **Disk**, which implements the Elevator and FIFO disk scheduling policies; a **Failure Manager**, which is responsible for initiating the failure mode of operation; and a **Log Manager**, which collects and stores undo/redo log records.

Figure 5.1 illustrates the overall architecture of our performance model. Each component described above is represented as a circle in Figure 5.1, and is implemented as a DeNet[Livn89] discrete event module. The arcs in the figure are discrete event connectors. Each arc can be thought of as a combination of a preconstructed message path and a set of predefined message types. The following sections present a more detailed description of the functionality of each component in the model.

5.2. The Database Manager

The database used in the simulation is modeled as a collection of relations (files). Each file, in turn, is modeled as a collection of data pages. Indices, including both clustered and nonclustered B+ trees, can be built on top of a file. The Database Manager initializes the database and distributes the tuples of a relation to disks (i.e. constructs the mapping between the logical tuple ids of a relation and their physical disk addresses.) Table 5.1 summarizes the key parameters of the Database Manager. The number of relations in the database is *NumRels*. For each relation i ($1 \le i \le NumRels$), $DegDecl_i$ is the degree of declustering [Cope88] of relation i, $RelSize_i$ is the relation size in tuples, $TupleSize_i$ is the tuple size in bytes, and $NumIndex_i$ is the number of indices associated with relation i. If $NumIndex_i$ is 0, relation i has no indices. If $NumIndex_i$ is greater than 0, index number 1 is always the clustered index, while the remaining index numbers, if any, represent nonclustered indices. For each index j ($1 \le j \le NumIndex_i$), $KeySize_{i,j}$ is the size of the key attribute of index j, and $KeyPtrSize_{i,j}$ is the size of a tuple pointer in an index record. $KeySize_{i,j}$ and $KeyPtrSize_{i,j}$ together with *DiskPageSize* determine the fanout of an index node. In turn, the fanout parameter together with the RelSize determine the depth(s) of the B+ tree(s).

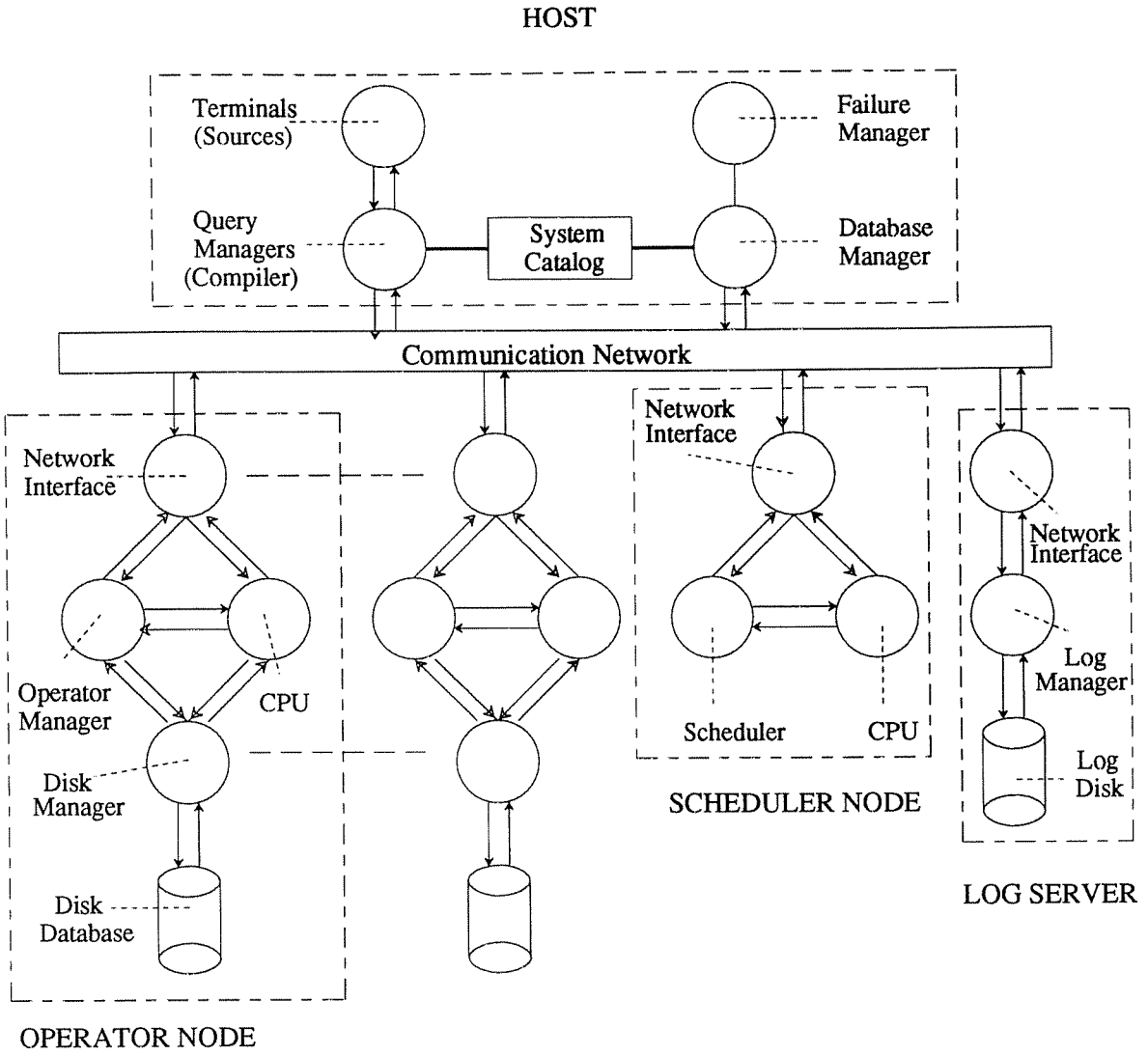

Figure 5.1: Architecture of the Simulation Model.

Parameter	**Meaning**
NumNodes	Number of nodes in the experiment
NumRels	Number of relations in database
$DegDecl_i$	Number of disks relation i is distributed over
$StrtDisk_i$	Disk number that stores the first tuple of relation i
$RelSize_i$	Number of tuples in relation i
$TupleSize_i$	Tuple size in bytes of relation i
$NumIndex_i$	Number of B+ tree indices relation i has
$KeySize_{i,j}$	Size of key attribute of index j of relation i
$KeyPtrSize_{i,j}$	Pointer size in an $index_j$ key record of relation i
DiskPageSize	Size of a disk page in bytes
$RespFracPri_{i,k}$	Responsible fraction of k-th primary fragment of relation i
$RespFracBkup_{i,k}$	Responsible fraction of k-th backup fragment of relation i

Table 5.1: Database Model Parameters.

All of the database information described in the previous paragraph is stored in the system catalog. In addition, when data is replicated, the location of the primary and backup fragments of each relation and their responsible ranges (fractions) are also stored in the system catalog. The responsible fraction is initialized to 1.0 for all primary fragments and 0.0 for all backup fragments. When a disk or processor failure occurs, the responsible fraction of each fragment is recomputed using the formulas described in Section 3.5 and the system catalog is updated accordingly.

5.3. The Terminal and Query Manager Modules

The Terminal module is the component responsible for generating queries (i.e., the workload). Table 5.2 summarizes the key parameters of this module. *QueryType* models the type of the current query. A query request can be one of the following three types in this study: select to host, select to file, or update. *AccessPath* indicates which access method will be used by the query to retrieve data pages from the disk. *QualAttr* is used to indicate whether the qualified attribute of a query is also a range (or a hash) partitioning attribute[2]. In this case, the query will potentially be sent to only a subset of the nodes for processing. Otherwise, the query will be sent to all nodes. A query can be executed by either a sequential file scan or by using a clustered or a nonclustered index. *IndexId* indicates the identity of the index to use when an indexed access method is selected. *RelationId* and *TupleId* for indexed accesses are each randomly selected using a uniform random number generator. A query may select or update any number of tuples. This is captured in the *Selectivity* parameter which can range from 0%, which selects (updates) no result tuples, to 100%, which selects (updates) all of the tuples in a relation. The model simulates a closed system, so there can be only one outstanding request per Terminal; the number of Terminals is modeled by the *MultiProgLevel* variable. To model a single user environment, this parameter is set to 1. When a query is completed, a Terminal waits for exactly *ThinkTime* seconds before submitting another query. Finally, the simulation runs until the preselected response time confidence interval, *ConfidInt*, is reached.

Given a query request, the Query Manager module examines the schema information and uses the Responsible Range table to determine the responsible node(s) for the query. It then constructs a query plan tree. Depending on the mode of operation (normal or failure), the Query Manager may generate a different query plan for the same query request. If a single node is responsible for the query, the query plan constructed will be sent directly from the Query manager to the responsible node. Otherwise, the query plan will be sent to the Scheduler.

Parameter	**Meaning**
QueryType	Type of query
AccessPath	Access path used to access a relation
QualAttr	Qualified attribute
IndexId	Index structure used to access a relation
RelationId	Relation to access for the current query
TupleId	Starting tuple id for an indexed access method
Selectivity	Fraction of a relation satisfying selection criterion
MultiProgLevel	Number of terminals that generate requests
ThinkTime	Think time for the terminals
ConfidInt	Response time confidence interval

Table 5.2: Workload Model Parameters.

5.4. The Scheduler Module

The Scheduler module is responsible for coordinating the execution of queries that execute on multiple processors. For each query plan tree received, the Scheduler traverses the tree top down; for each vertex in the tree, it sends a message to activate an operator process on each of the responsible nodes. When a node failure has occurred, the vertices of a query tree may be modified by the Scheduler, if necessary, before it is sent to the set of responsible nodes for processing. After initiating the execution of a query, the Scheduler waits for an "operator done" message from each operator process. Once all of the operator processes corresponding to a query have replied, the Scheduler commits the query and sends a "query done" message to the requesting Terminal.

5.5. The Network Manager and Interface Modules

The network manager encapsulates the communication network. A key parameter of this module is *PacketThreshold*, which determines how many network packets can be served simultaneously. Network packets are served in first-come, first-served (FCFS) order by the Network Manager. When a packet arrives at the Network Manager, it is served immediately if there are less than PacketThreshold packets outstanding. Otherwise, the new packet will be placed in a queue; as soon as a packet leaves the Network module, the head of the waiting queue will be removed and service for it will begin. While being served, each packet is delayed for a time T in the network module before it is delivered to the destination node. T is proportional to the number of bytes in the packet, which includes a packet header. The size of a network packet ranges from several hundred bytes for a control packet to several thousand bytes for a data packet.

The Network Interface module models the sending and receiving of network packets (messages) for an operator node. For each message sent and received, a certain amount of CPU resource will be consumed. The amount of CPU time required for packaging and sending a packet is determined by the message type (e.g. data or control packet) and the size of the message.

5.6. The Operator Manager Module

The Operator Manager simulates the operator processes of the Gamma database machine. This module models two different types of operator processes: selection processes and store processes. Table 5.3 summarizes the key parameters of the Operator Manager module. Upon receiving a new query packet, the Operator Manager executes *OpStartupCPU* instructions to start up a new operator process. The newly activated operator process then decodes the incoming query packet. Depending on the type of the incoming query packet, the operator process may begin requesting data pages from the Disk Manager (if it is a select) or it may wait for a data packet to arrive from another processor via the network module (if it is a store). Each operator process will require some CPU processing time when it initiates an I/O request and when it processes disk or network data pages. Different types of operations will require different numbers of CPU processing instructions. *IndexRecCPU* specifies the number of instructions required to process an index record, and *SelTupleCPU* specifies the number of instructions required by a selection operation to process a tuple. *ByteCopyCPU* specifies the number of instructions needed to copy a byte in memory for the store operation, while *DiskWriteCPU* specifies the number of instruction needed to initiate a disk write operation. Finally, when an operator process completes execution, *OpdoneCPU* instructions are consumed to deactivate the process.

Parameter	**Meaning**
OpStartup	Number of CPU instructions to start up an operator process
IndexRecCPU	Number of CPU instructions to process an index record
SelTupleCPU	Number of CPU instructions to process a tuple for a select
ByteCopyCPU	Number of CPU instructions to copy a byte within memory
DiskWriteCPU	Number of CPU instructions to initiate a disk write operation
OpDoneCPU	Number of CPU instructions to deactivate an operator process
BufHitRatio	Percentage of buffer hits

Table 5.3: Parameters of the Operator Manager.

5.7. The CPU Module

The CPU module models the sharing of the CPU resource among different processes for a node. When a process needs CPU cycles, it sends a request to the CPU module with the number of CPU instructions needed. If the CPU is free, the request is served immediately and a reply is sent back to the requester after the requested CPU time has elapsed. Otherwise, the request will be put in a CPU ready queue. There are two types of FCFS queues in the CPU module: a high priority queue and a low priority queue. CPU requests generated by the Disk Manager for transferring data from the I/O channel's FIFO buffer to main memory are put in the high priority queue, whereas requests from the Operator Manager or the Network Interface are put in the low priority queue. Requests in the low priority queue are served by the CPU when the high priority queue is empty. If a high priority request arrives while a low priority request is being served, the low priority request is preempted and the high priority request receives CPU service immediately. The CPU resumes service of the preempted request when the high priority queue is once again empty. A key parameter of the CPU module is the CPU speed in *MIPS*.

5.8. The Disk Manager and Disk Modules

The Disk Manager is responsible for handling I/O requests generated by the Operator Manager. When a disk request is received, the Disk Manager maps the logical page number generated by the Operator Manager to a physical disk address (cylinder #, sector #), issues a disk I/O request, and then waits for the completion of the disk request. When more than one disk is attached to a processor, the Disk Manager determines the responsible disk based on the global disk id information associated with the request.

When data is replicated, the simulation model assumes that the primary copy access scheme is to be used for both chained declustering and interleaved declustering. In the normal mode of operation, all read requests are served by the disk with the primary copy, whereas write requests are sent to the disk storing the primary copy and the disk storing the backup copy. With mirrored disks, a disk read is served by the disk of the mirrored pair that has the shortest seek time. Disk writes are sent to both disks in the mirrored pair and the physical writes to two disks are synchronized.

The Disk module employs an elevator disk scheduling discipline except in the case of mirrored disks. In this case, a FIFO (with the shortest seek time) scheduling discipline is used [Bitt88, Gray88]. The total time required to complete a disk access is modeled by the following formula:

Disk Access Time = Seek Time + Rotational Latency + SettleTime + Transfer Time

The Seek Time for seeking across n tracks is modeled by the formula [Bitt88]:

Seek Time(n) = $SeekFactor * \sqrt{n}$

The Rotational Latency is modeled by a random function which returns uniformly distributed values in the range of *MinLatency* to *MaxLatency*. *SettleTime* models the disk head settle time after a disk arm movement. The value of Transfer Time is computed by dividing the disk page size by the disk *XferRate*. Table 5.4 summarizes the key parameters of the physical resource aspects of the model (CPU, disk, and network).

Parameter	**Meaning**
MIPs	CPU speed
SeekFactor	Used to compute disk seek time
MinLatency	Minimum rotational latency of disk
MaxLatency	Maximum rotational latency of disk
SettleTime	Disk head settle time
XferRate	Transfer rate of disk
PacketThreshold	Number of simultaneous packets for network

Table 5.4: Resource-Related Parameters of the Model.

5.9. The Failure Manager and Log Manager Modules

The Failure Manager has no impact when the normal mode of operation is simulated. When simulating the failure mode of operation, this module will randomly select a node to fail and will recompute the responsible range of each fragment (the fraction of responsibility, not the actual attribute range). The newly computed responsible ranges are given to the Database Manager, which in turn updates the responsible range table stored in the system catalog. New query plans will be generated when required based on this new information.

The Log manager is not actually implemented. However, because all of the schemes simulated here have approximately the same overhead for generating and storing log records, we believe that the exclusion of the Log Manager will not significantly affect the relative performance of the schemes.

5.10. Example of a Query Execution

To make the interactions between different modules in the simulation model more clear, we describe in this section how a selection query is executed in our model. At simulation time T, a new request is generated at the Terminal module. The request is then forwarded to the Query manager. The Query manager creates a query plan based

on the request and on the information stored in the Schema and the Responsible Range table. If a single node is responsible for the current request, the corresponding query packet will be sent directly to the query processing node (Operator node). Otherwise, the query packet will be sent to the Scheduler, and the Scheduler will initiate a multi-node transaction by sending the query packet to all responsible Operator nodes. Assume in this example that the query is sent to several nodes for processing and that node 3 is one of the nodes responsible for the query. After some communications (network) delay, the query packet arrives at the Network Interface (NI) module of node 3. Upon receiving the packet, The NI module sends a message to the CPU module of node 3 requesting NPP (Network Packet Processing) units of CPU time. After NPP time units, the CPU module replies to the NI module. Upon receiving the reply, the query packet will be forwarded from the NI module to the Operator Manager (OM) module.

One of the operator (select) processes in the OM module is activated to process the incoming query packet. The newly activated operator process will decode the query packet, and retrieve corresponding tuples from the local disk through an index structure or using a file scan. For every requested page (index or data page), the operator process initiates a disk I/O request to the Disk Manager module, and then wait for the page to become available. After some disk service delay, the Disk Manager sends a request to the CPU module requesting the transfer of data from the FIFO buffer to main memory. After FTMT (FIFO To Memory Transfer) units of time, the CPU module informs the waiting operator process that data is available for processing. Depending on the type of query, the operator process will send a message to the CPU module to request PP (Page Processing) units of CPU time. After PP units of time delay, the CPU replies back to the operator process indicating that the page processing has been done. Matching tuples (if any) are returned to the Terminal module in the unit of a predefined network data packet.

Disk I/O requests and page processing at the CPU module may iterate several times if a subquery accesses more than one disk page. When the processing of a query at an Operator node is done, the OM module sends an "operator done" message to the Scheduler. Upon receiving "operator done" messages from all participating OM modules, the Scheduler commits the query and informs the waiting Terminal module of the outcome. Upon receiving a "commit" message from the Scheduler, the Terminal module creates a new query and the query processing cycle repeats.

5.11. Physical Data Placement of the Primary and Backup Copies

Figure 5.2 illustrates the layout of the primary and backup fragments on four disks for the mirrored disks, chained declustering, and interleaved declustering schemes. As illustrated in Figure 5.2, relation R is partitioned across four disks (or two mirrored pairs). R_i is the i-th primary fragment of relation R, while r_i is the corresponding backup fragment. With the mirrored disk strategy, the contents of the two disks within a mirrored pair are identical (R1+R2 = r1+r2 and R3+R4 = r3+r4). R_i+R_j in the mirrored disk strategy represents the union of fragments R_i and R_j. With chained declustering, for a given disk drive, primary fragments with their associated indices from all relations are placed together on the outer half of the cylinders while backup fragments are stored on the inner half. With interleaved declustering, primary fragments and their associated indices from all relations are placed together on the outer half of a disk drive (as with chained declustering) while all backup subfragments are placed together on the inner half.

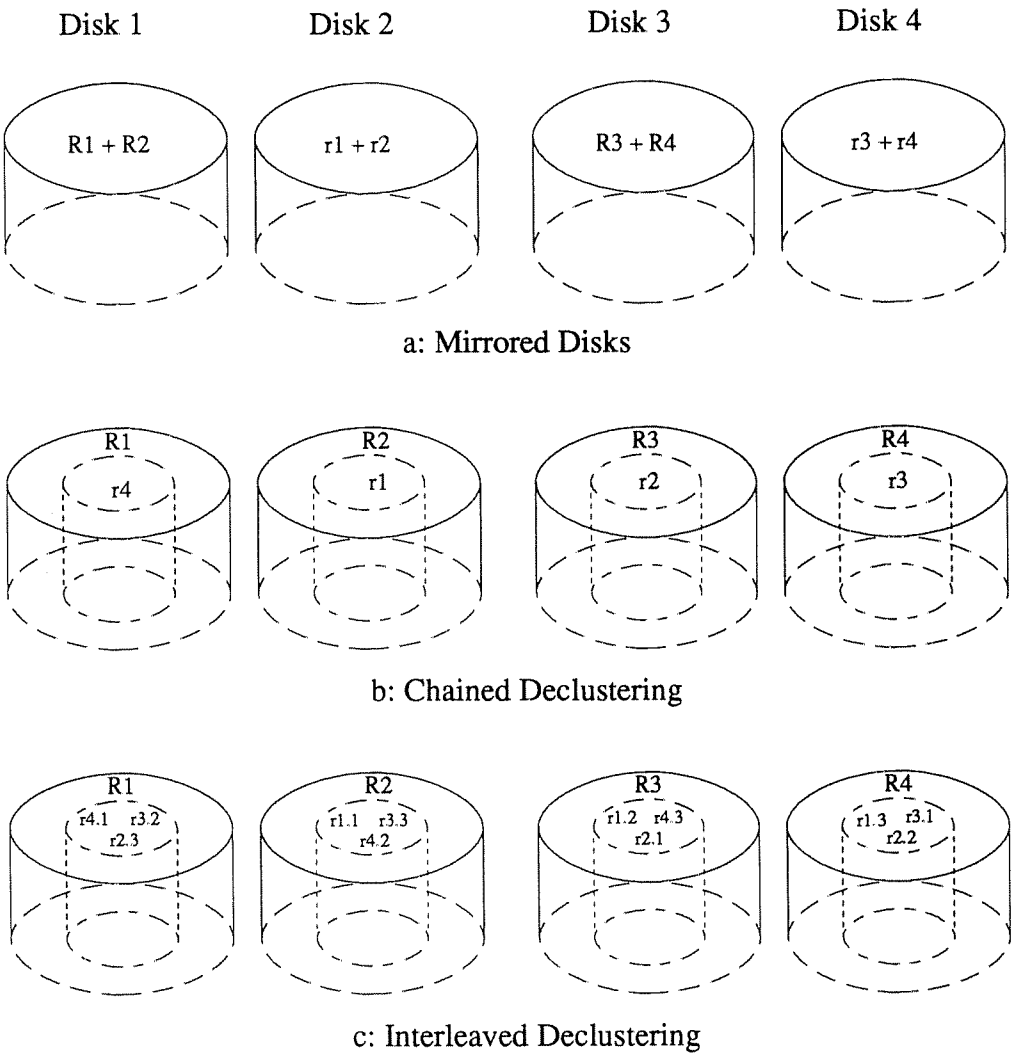

Figure 5.2: Data Placement

With the mirrored disk scheme, a disk read request can be served by either disk in the pair. In Tandem's system, and in our experiment, the disk with the shortest seek time is assigned to serve a disk read request. As a result, the expected seek distance for random reads is one-sixth of the tracks [Bitt88, Gray88] (instead of one-third as in the case of a single disk). With both chained declustering and interleaved declustering, the primary fragment access scheme is used in our simulation experiments. Because primary fragments are placed together on the outer half of a disk drive, the expected seek distance for random read requests is also reduced from one-third to one-sixth of the tracks. As a result, all three data replication schemes provide improved performance over the non-replicated case for read only queries.

When we first implemented the simulation model, all data files were stored adjacent to each other on disk and all index files were stored together. The rationale behind this data placement was that a DBA might load all data files before any indices are created. Although this data placement strategy works fine for the other two schemes, it results in a significant problem for the mirrored disk scheme. With mirrored disks, a disk read request is served by a disk, within a mirrored pair, that has the shortest seek distance. For selection queries using an index to retrieve data pages, one disk in a mirrored pair will handle all disk reads from the index files while the other disk will be responsible for all reads from the data files. Unless the number of index and data pages accessed are equal, one disk will serve more requests than the other. If the numbers of page reads are significantly different, the workload in the mirrored pair will be skewed and the performance of the mirrored disk strategy will degrade significantly. In order to be fair to the mirrored disk strategy, we modified the original data placement algorithm to avoid this problem by placing an index file adjacent to the data file it is indexing.

5.12. Disk Scheduling Algorithms

As mentioned in Section 5.8, the elevator disk scheduling algorithm is used with both the chained declustering and interleaved declustering schemes. With the mirrored disk scheme, on the other hand, a read request is directed to the disk with the shortest seek time, and a disk write request writes to both disks in a mirrored pair simultaneously (with no CPU overhead for initiating an additional disk write request). In order to receive the benefit of shortest seek time for read requests and to perform synchronized disk writes, disk requests in a mirrored pair are served in the order of their arrival both in the Tandem system [Gray88] and in our experiment.

5.13. Remote Update Policy

With chained declustering and interleaved declustering, the primary and backup copies of a tuple are stored on two disks attached to two different processors. An update query in these two schemes can be processed in one of the following three ways. First, an update query could be directed to nodes storing either the primary or backup copies of the matching tuple(s) for processing. Each node would then retrieve the tuples from disk, update them, and write them back to disk. However, processing an update query at both the primary and backup copy nodes, an update query would access twice as many disk pages as in the non-replicated case. For example, assuming that the depth of an index structure is 3, a single tuple update query on the partitioning attribute using an index will read 4 pages and write 1 page in the non-replicated database whereas, with replication, it will access 5 disk pages on each

of the two nodes storing the matching tuple (for a total of 10 disk page accesses).

Instead of sending an update query to both the primary and backup copy nodes, the second method directs an update query only to the primary copy node for processing. After the processing at the primary copy node is completed, update records (redo log records) are gathered and sent from the primary copy node to the backup copy node. To update the backup copy, the backup copy node decodes the update records and reads the corresponding tuples (pages) from the disk. The tuples (pages) read are then updated using the update records and written back to disk. Given the above single tuple update query, the backup copy node will read only one page and write one page with this method. As a result, the single tuple update query will read 5 pages and write 2 pages resulting in a saving of 3 disk accesses over the first method.

The last method also directs an update query only to the primary copy node for processing. However, instead of sending update records, the primary copy node sends the updated pages to the backup copy node. Upon receiving an updated page, the backup copy node initiates a disk write operation and writes the page to disk. With this method, there will be additional communications cost but fewer disk pages will be accessed than with the second method. Because the network wire delay is 5.6 ms for an 8K data page (measured on the Intel Hypercube), while the average disk service time for read requests is more than 12 ms, we selected the third method for processing update queries with chained and interleaved declustering in our simulation experiments.

CHAPTER 6

PERFORMANCE RESULTS

6.1. Introduction

In this chapter, we present the results of our simulation experiments to study the performance of three identical copy based high availability strategies, namely the chained declustering (CD), interleaved declustering (ID), and mirrored disk (MD) strategies. The objective of this study is to analyze the relative performance of these three high availability schemes under different workload and operation mode assumptions. As demonstrated in Chapter 4, chained declustering is able to provide the best load balancing when a node failure occurs while mirrored disks provides the worst. In this chapter we investigate, through simulation experiments, how the load balance/imbalance affects system throughput and response time in various situations. Because a system is expected to operate in the normal mode more frequently than in the failure mode, a high availability strategy will not be very useful if it cannot also perform well in the normal mode. To verify that the chained declustering scheme can perform well in this mode, we also study the relative performance of the three strategies when no failures have occurred. Besides comparing the performance of the different high availability strategies, two other related issues are explored in this chapter as well: the performance impact of updating backup copies on the different high availability strategies, and the tradeoff between the benefit of intra query parallelism and the overhead of activating and scheduling extra operator processes.

The remainder of this chapter is organized as follows. We begin by validating (in Section 6.2) the simulation model using results obtained from the Hypercube version of the Gamma database machine [DeWi90]. We then discuss the performance metrics of interest and the parameter settings used in Section 6.3. In Section 6.4 we present and analyze the results of our performance experiments. Finally in Section 6.5, we summarize our results and present our conclusions.

6.2. Model Validation

To evaluate the reasonableness of the simulation model, we first configured the model with one disk per node and ran a number of experiments without data replication. The results obtained from these experiments were then validated against results produced by the Gamma database machine.

Tables 6.1 gives the values of the key parameter settings in our validation experiments. In the validation experiments, the model is configured to have from 5 to 30 nodes. Each node has a processor (CPU), with one disk attached to each processor. The database consists of 2 relations, each containing 1 million tuples of data. Both relations are fully declustered over all available disks. The size of each tuple is 208 bytes, and two indices per relation are constructed (one clustered and the other nonclustered). The size of a disk page in the validation experiments is set to 18K bytes. The settings for CPU related parameters are as follows: The number of instructions to start up an operator process at an operator node is 10,000. Selecting a tuple (including predicate application) takes 400 instructions, while storing a tuple takes 300 instructions. Copying 1 byte from memory to a network buffer or vice versa takes 1 CPU instruction. Initiating a disk write for an updated page takes 2000 instructions, while deactivating an operator process takes 1000 instructions. For disk parameters, the seek distance is measured by the distance between the current location (track number) of the disk arm and the location of the current request. As explained in Section 5.2, seek time is computed based on the square root formula described in [Bitt88]. The seek factor used to compute seek time is set to 0.78, which was derived from the data sheet provided by the disk manufacturer for the disks used by the Gamma database machine. Minimum and maximum rotational latency are set to 0 ms and 16.667 ms, respectively. The rotational latency for each individual request is randomly (uniformly) generated between these two values. The data transfer rate from disk to the disk buffer is 2M bytes per second.

Parameter	Setting
NumNodes	5 - 30 nodes
DiskpNode	1
NumRels	2 relations
$DegDecl_i$	5 - 30
$RelSize_i$	1M tuples
$TupleSize_i$	208 bytes
$NumIndex_i$	2
$KeySize_{i,j}$	4 bytes
$KeyPtrSize_{i,j}$	8 bytes
DiskPageSize	18K bytes
Selectivity	single tuple and 1%
MultiProgLevel	1
ThinkTime	0 sec
ConfidInt	within 5% (95% confidence)
OpStartup	10,000 instructions
IndexRecCPU	50 instructions
SelTupleCPU	400 instructions
StoTupleCPU	300 instructions
ByteCopyCPU	1 instruction
DiskWriteCPU	2000 instructions
OpDoneCPU	1000 instructions
MIPS	3
SeekFactor	0.78
MinLatency	0 msec
MaxLatency	16.667 msec
SettleTime	2.0 msec
XferRate	2M bytes/sec
PacketThreshold	999

Table 6.1: Parameter Settings for Validation Experiments.

Three query types are used in the validation process: a single tuple selection query using a sequential file scan, a 1% selection query using a nonclustered index, and a single tuple selection query using a clustered index. The first two query types store their results on disk using a round robin partitioning strategy. The third query type returns its results to the user. The results of two of the three query types are shown in Figures 6.1 and 6.2. Figure 6.1 shows the performance of a single tuple selection query using a sequential file scan (nonindexed), and Figure 6.2 shows the performance of a 1% selection query using a nonclustered index (nclidx). In these two figures, the abscissa is the number of nodes (disks) in the experiments and the ordinate is the average response time of the query in seconds. As shown in Figures 6.1 and 6.2, the results produced by the simulation model accurately match Gamma's actual performance for both of the query types. The difference in performance between the model and Gamma is less than 10% in these two figures. The third query type, whose performance result is not shown here, also produced a close match between the model and Gamma.

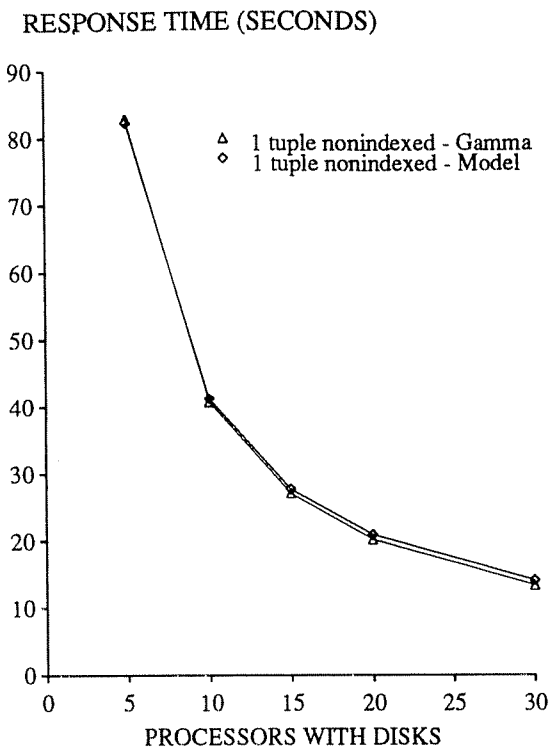

Figure 6.1

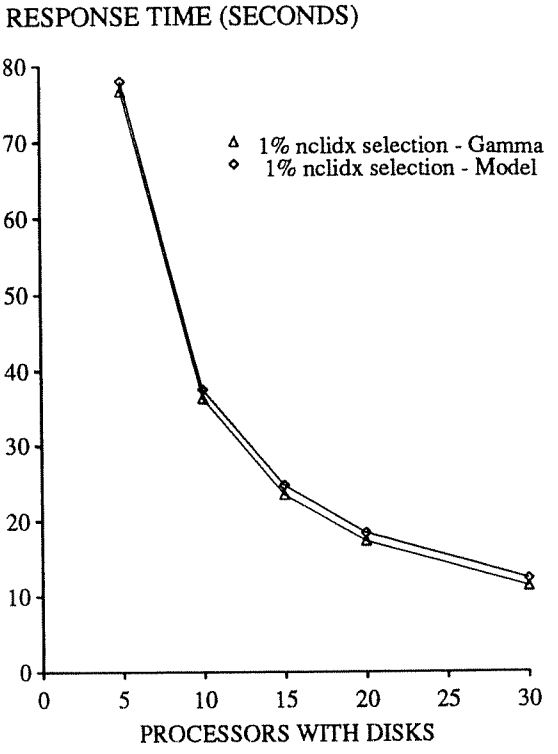

Figure 6.2

6.3. Performance Metrics and Parameter Settings

In most computer systems, system throughput and average job (query) response time are two of the major performance metrics that people are interested in. Because our model simulates a closed system, response time is inversely proportional to system throughput in our performance results. Throughout this chapter, system throughput will be used as the main performance metric for comparing the performance of different strategies. Besides system throughput, several additional metrics will be used to aid in the analysis of the experimental results. The first one is the *disk service time*, which is the average time to serve a disk I/O request. This metric is measured from the beginning of disk service (disk arm movement on the behalf of current request) to the completion of the current request (transfer of last byte completed). The disk service time metric does not include the time spent waiting in the disk queue. The second metric is the *disk utilization*, which is computed by dividing the total disk service time of a disk by the experiment time. The third metric is the *CPU utilization*, which is measured by dividing the total CPU busy time by the experiment time. Finally, the average number of index and data pages accessed per query is also examined in our experiments.

In our performance experiments, the parameter settings are the same as in the validation experiments except for the following (as indicated in Table 6.2). The number of nodes (processors) in our performance experiments is set to 16. Because the mirrored disk scheme requires that at least two disks (a mirrored pair) are connected to a node, two disks are attached to each processor in the subsequent experiments. With 16 processors and two disks attached to each processor, a total of 32 disks are used in the simulation model. The number of relations is 8, each containing 2 million tuples. Relations are fully declustered across all available disks. The number of terminals (sources) in the model ranges from 1 to 288, and each terminal can have at most one outstanding request. The buffer hit ratio for disk read requests is assumed to be 20%. The CPU speed is set to either 3 MIPS or 14 MIPS in this study. We picked these two CPU speeds because the processor of a Gamma node (which is an Intel 80386) is rated at about 3, while the current generation of microprocessors are rated at around 14[1]. The cluster size (number of disks) of the interleaved declustering scheme is set to 8 because Teradata recommends a cluster size to 4 or 8 to its customers.

Parameter	Setting
NumNodes	16
DiskpNode	2
NumRels	8 relations
$RelSize_i$	2M tuples
$DegDecl_i$	32
MultiProgLevel	1 - 288
BufHitRatio	20%
MIPS	3 and 14
IDclusterSize	8
DiskPageSize	4K and 8K bytes

Table 6.2: Parameter Settings for Performance Experiments.

The size of a disk page is set to either 4K or 8K bytes in the performance experiments, whereas it was set to 18K bytes in the model validation experiments. This is because 4K and 8K page sizes are commonly used in current database systems. The reason for validating our model with an 18K page size is because we have a full set of results produced by Gamma using an 18K page size. We obtained these results with the hardware read ahead feature[2] turned off. Instead of implementing a hardware read ahead feature in our model or disabling the hardware

[1] Intel 80486, Motorola 68040, Sun SPARC chip set, and the MIPS R2000 are all rated between 12 and 15 MIPS.

[2] The disk drive in the Intel Hypercube has a hardware read ahead feature which tries to read 2 and a half tracks of data into the disk cache while serving a disk page request. If the next request can be satisfied with data from the disk cache, the data will be returned directly from the cache and no physical disk access will be required.

read ahead on Gamma and then validating our model for either the 4K or 8K disk page, we elected to validate our model using an 18K disk page.

6.4. Performance Results

The presentation of the performance results is organized as follows. We first study the relative performance of the chained declustering (CD), interleaved declustering (ID), and mirrored disks (MD) strategies using selection (read only) queries. We then investigate the performance of these three data replication schemes using workloads that require both disk reads and writes. Finally, we look at how varying the CPU speed and disk page size parameters affects the relative performance of the three replication schemes.

Because the Intel 80386 processor used in the Hypercube version of Gamma is rated at 3 MIPS, and Gamma is currently configured to use a disk page size of 8K bytes, we will first present our performance results with the page size and CPU parameters set to 8K bytes and 3 MIPS, respectively. After that, we will present results with different combinations of CPU and page size parameters. In the following, unless noted otherwise, the results shown are obtained with 3 MIP CPUs and 8K byte disk pages.

6.4.1. Selection Queries (Experiments 1 - 3)

Three selection query types are studied in this section: a single tuple selection query on the partitioning attribute using a clustered or nonclustered index, a 0.1% selection query on a nonpartitioning attribute using a nonclustered index, and a 1% selection query on a nonpartitioning attribute using a clustered index. We assume that relations in the simulation experiments are either range or hash partitioned. The first query type (single tuple selection query) will be sent to a single node for processing[3], while the other two query types (0.1% and 1% selection queries) will be sent to all nodes. All three selection query types return the matching tuple(s) back to users/application programs through the communication network.

Experiment 1: Single tuple selection on the partitioning attribute

As described in Chapter 5, in order to reduce the number of disk pages accessed for updating a backup copy at the node storing the backup copy, an updated page is sent from the primary copy node to its backup copy node.

[3] This is true for all but the ID scheme, where in the event of a disk failure a single tuple selection query on the partitioning attribute may need to be sent to multiple backup copy nodes for processing.

Upon receiving an updated page, a backup node will initiate a disk write operation and write the page to disk. With the CD and MD strategies, this can be done very easily because the primary and backup copies of a fragment are identical (the i-th page in a primary copy is identical to the i-th page in its corresponding backup copy). With the ID strategy, however, this is not necessarily true. As described in Chapter 2, with the ID scheme a backup copy is sub-divided into multiple subfragments and each subfragment is stored on a disk other than the one containing the primary copy. To make this page update (update propagation) strategy possible, backup fragments with ID must be divided based on page number rather than on attribute values. For example, to divide a backup fragment with 15 pages into 3 subfragments, the first 5 pages will be stored in subfragment 1, the next 5 pages (page 6 through 10) in subfragment 2, and the last 5 pages in subfragment 3. However, when dividing a backup fragment based on pages, a query originally served by a single failed primary fragment may need to be sent to multiple backup subfragments for processing. For example, suppose that a relation is hash partitioned on attribute X across a set of disks and that the clustered index is also constructed on X. Within a primary fragment, tuples will be stored according to X attribute values. If the corresponding backup fragment is divided into subfragments according to page numbers, each subfragment will most likely have the same hash value range (equal to the hash value range at the primary copy). When the primary fragment fails, a single tuple selection query originally served by the failed primary fragment, P, has to be directed to all of P's backup subfragments for processing. This will result in both CPU and disk overhead of activating extra operator processes and accessing extra index pages. The only case in which this complication doesn't occur is when relations are range partitioned and a clustered index is also constructed on the partitioning attribute (X). In this case, backup subfragments will have disjoint X values and a single tuple selection query on the partitioning attribute will be directed to a single node even when the responsible primary fragment has failed.

In a nutshell, in the event of a disk failure, the performance of a system that uses ID is greatly influenced by the partitioning method and the indexing scheme used. When a relation is range partitioned and a clustered index is constructed on the partitioning attribute, the query compiler and scheduler can generate a query plan and direct the plan to a single node for processing. With all other combinations, the query scheduler will have no knowledge about where the matching tuple(s) resides when the responsible primary fragment fails and the query has to be directed to all corresponding backup subfragments (even though only one of the backup subfragments may return the matching tuple(s)).

Case A: In this experiment, we assume that a relation is hash partitioned and an index (clustered or nonclustered) is constructed on the partitioning attribute[4]. Figure 6.3 shows the average throughput of a single tuple selection query on the partitioning attribute using an index. As shown in Figure 6.3a, all three schemes provide approximately the same level of throughput at low multiprogramming levels (MPL < 24). As described in Section 5.3, all three schemes have expected seek distance of one-sixth of the tracks for random reads in the normal mode of operation. With 32 disks and less than 24 outstanding disk requests, the chance that there is more than one request in a disk queue is very small, and disk requests are likely to be served in the order of their arrival with all three schemes. Consequently, the disk service times with all three schemes are approximately the same.

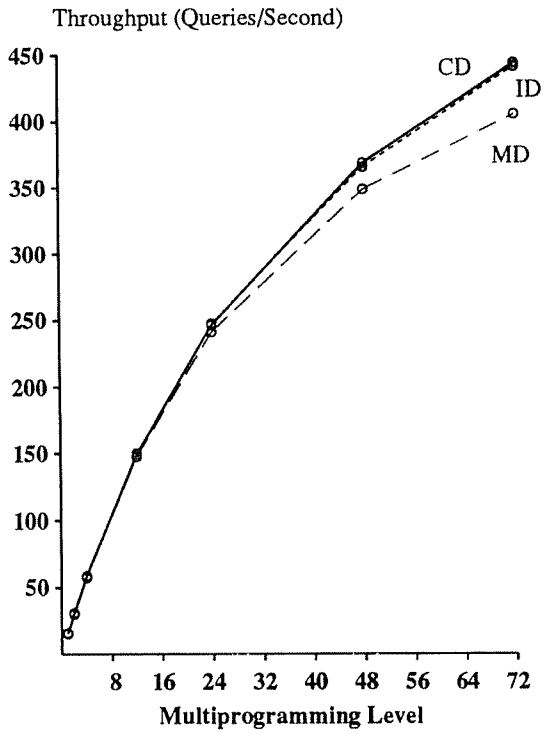

Figure 6.3a: Single Tuple Selection
Normal Mode

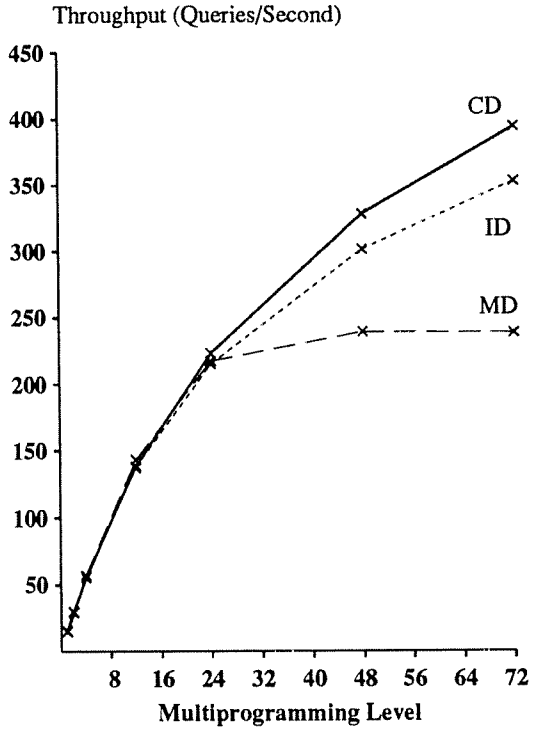

Figure 6.3b: Single Tuple Selection
Failure Mode

When the MPL is greater than 24, the probability of more than one request waiting in disk queue becomes higher. In turn, the effectiveness of the elevator disk scheduling algorithm used by the CD and ID schemes also increases. As a consequence, the average seek distance becomes smaller with the CD and ID schemes than with the

[4] If a relation is range partitioned and a nonclustered index is constructed on the partitioning attribute, the relative performance of the three schemes will be the same as is shown in this case.

MD scheme (which uses SSTF to select a disk and then FIFO within a disk to schedule arm movement [Gray88]). At a MPL of 72, for example (Figure 6.4), the average disk seek (service) time with CD is 7.19 ms (21.62 ms), whereas it is 8.31 ms (22.74 ms) with MD. As shown in Figure 6.3a, the systems that use CD and ID provide about the same level of throughput and they both process more queries per second than MD when MPL $\geq$ 48.

In the failure mode of operation, (Figure 6.3b) all three schemes suffers little (or no) performance degradation at low multiprogramming levels (MPL $\leq$ 4). This is because the systems (processors and disks) are under utilized when MPL $\leq$ 4. Although the workload increase at a disk due to failures of other disks results in an increase in disk utilization, the impact on throughput is very small here. As the MPL increases, however, the impact of a disk failure becomes more and more significant. When MPL > 24, the throughput of the MD scheme levels off because the remaining disk in the failed mirrored pair is fully utilized and becomes a bottleneck. This is because the MD scheme cannot balance the workload when failures occur. The remaining disk (disk #3 in our experiments) of a failed mirrored pair is responsible for all the disk requests that were originally served by two disks. Since selection queries are read only queries, the workload on the remaining disk increases by 100% in this case. With the CD and ID schemes, on the other hand, the throughput continues to increase beyond a MPL of 24. This is because these two schemes do a better job of distributing the workload originally served by the failed disk.

Comparing Figures 6.3a and 6.3b, one can see that, at a MPL of 72, the decrease in throughput due to a disk failure with CD is about 10%, while it is about 20% with ID. In contrast, the decrease in throughput with mirrored disks is higher than 40%.

Figure 6.4 shows the disk service time in the normal mode of operation. As indicated in Figure 6.4, the disk service time with the MD scheme is a little bit higher than the other two schemes (even at a MPL of 1). At a MPL of 1, the difference mainly results from the fragment size difference between the MD scheme and the other two schemes[5] as explained in Section 5.3. A single tuple selection query using the clustered index sequentially accesses two index pages and one data page. Because a data fragment with MD is twice as large as it is in CD and ID, the seek distance from the second index page to the data page is longer with the MD scheme than it is with the other two schemes. As a result, the average disk seek time with MD is higher than the other two schemes at low multiprogramming levels. The difference in fragment size has little or no effect at high multiprogramming levels, however.

[5] Each fragment with the MD scheme is stored as two fragments in CD and ID, so each (primary) fragment in CD and ID is only half the size of that in MD.

When a disk head is serving requests from more than one query simultaneously, it will not serve all requests from the same query consecutively and thus the disk scheduling discipline becomes the determinant of the disk seek (service) time. The disk service time difference at high MPLs, as shown in Figure 6.4, is due mainly to the different disk scheduling disciplines used in the different schemes as explained above.

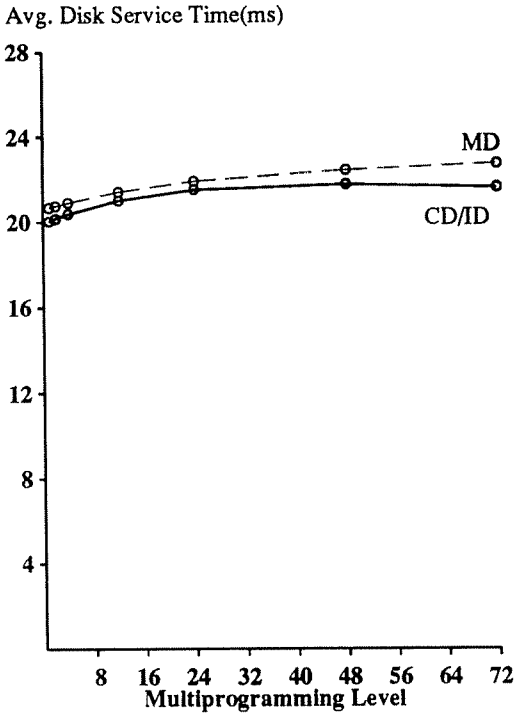

Figure 6.4: Single Tuple Selection, No Failure

Figures 6.5 and 6.6 show the disk and CPU utilization in the event of a disk failure. Figure 6.5 shows the utilization of each disk at a MPL of 4 and 72, and Figure 6.6 shows the utilization of each processor at the same multiprogramming levels. From these two figures, it is apparent that the single tuple selection query using a clustered index is I/O bound. As shown in Figure 6.5, the utilization of disk #3 (when disk #4 fails[6]) is much higher than that of the other disks with the MD scheme. For example, at a MPL of 4 the utilization of disk #3 is about 100% higher than the remaining disks in the failure mode. However, disk #3 is not fully utilized at this point, thus the throughput of the system continues to increase when the MPL is pushed higher. At a MPL of 24, it turns out that disk #3 is about 90% utilized. When the multiprogramming level is higher than 24, the load increase due to the failure of disk

[6] Disks 3 and 4 form a mirrored pair in our experiments.

#4 results in a bottleneck forming at disk #3. As a result, the throughput with MD levels off at MPL = 24 in the failure mode of operation.

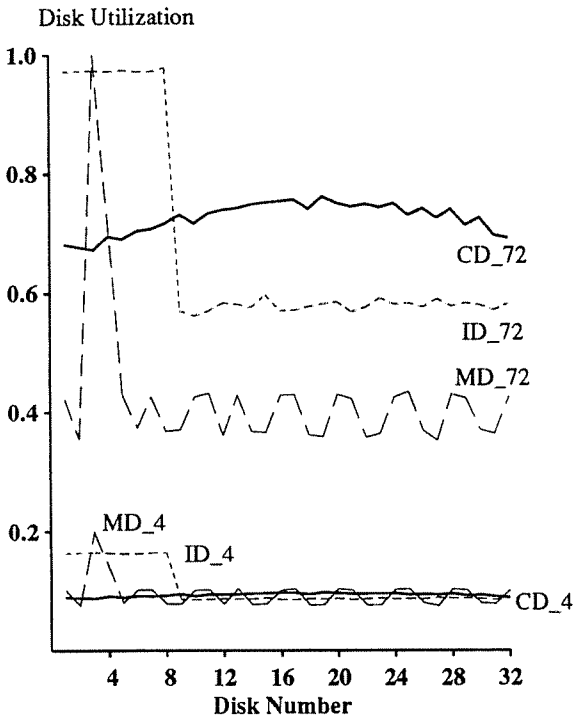

Figure 6.5: Single Tuple Selection Failure Mode

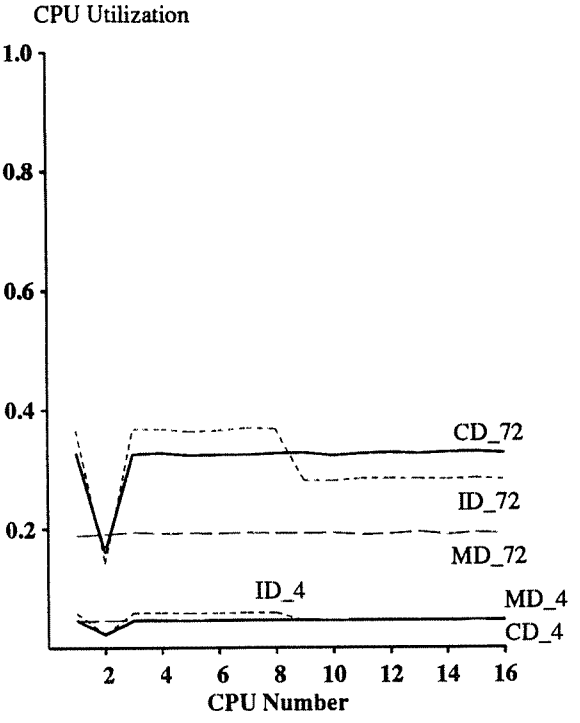

Figure 6.6: Single Tuple Selection Failure Mode

With the ID scheme, the utilization of the remaining disks in the failed cluster (disks 1 to 8) is also higher than that of the disks in the normal clusters (disks 9 to 32). At a MPL of 72, for example, the utilization of the remaining disks in the failed cluster is greater than 95%, whereas the utilization of the disks in the normal clusters is about 60%. However, as explained in the MD case, before a disk bottleneck is formed, the throughput of the system continues to increase when the MPL is pushed higher. With the CD scheme, on the contrary, the utilization of all disks is approximately the same throughout the entire experiment. At a MPL of 72, for example, the utilization of each of the remaining disks is around 70%. This is because, with the chained declustering scheme, the workload can be evenly redistributed among all remaining disks when a disk failure occurs.

Figure 6.6 shows the utilization of each processor in the failure mode of operation. As shown in the figure, with the MD scheme, the CPU utilization is about the same at all nodes. This is because, with MD, the workload does not shift from one node to the other in the event of a disk failure; all active processors have an equal number of

selection processes and process approximately the same number of pages. With the CD and ID schemes, on the other hand, the processor with the failed disk attached (processor #2) has a much lower utilization than the other processors. The reason is that the utilization of a processor is proportional to the number of pages processed which, in turn, is proportional to the number of operational disks attached to it. Since node (processor) 2 has only one operational disk, and the workload of the operating disks in the failed cluster is about the same, its CPU utilization is about 50% lower than the other nodes. Also, as shown in the figure, the utilization of the processors in the failed ID cluster (except processor 2) is higher than that of the remaining processors, whereas the utilization of all processors except #2 is about the same with the CD scheme.

Case B: In this experiment, we assume that the relation having the matching tuple is range partitioned AND the clustered index is also constructed on the partitioning attribute. With the CD and MD schemes, the results of this case are identical to that of case A above. With the ID scheme, the results in the normal mode of operation are also the same as they are in case A, but the results in the failure mode of operation are significantly different from that of case A. Unlike in case A, where a single tuple selection query on the partitioning attribute is directed to all the corresponding backup subfragments for processing when the responsible primary fragment fails, in this case (case B), the query can be directed to a single node for processing in both the normal and failure modes of operation. As a result, there is neither CPU processing overhead for activating additional operator processes nor disk I/O overhead for accessing extra index pages with the ID scheme .

Since the results in the normal mode of operation are identical to that of case A above, in the following we only show the results for the failure mode of operation. Figure 6.7 shows the average throughput of a single tuple selection query using a clustered index for the CD and ID schemes. As shown in Figure 6.7, both schemes provide approximately the same throughput at low multiprogramming levels (MPL < 12). This is because, as explained in case A, disks are under utilized in this region. When MPL > 12, the throughput with ID is actually higher than with CD. At a MPL of 72, for example, the ID scheme provides about 6% higher throughput than the CD scheme. At first, this result seems to contradict the conclusion given in Chapter 4 where CD is claimed to be able to provide better load balancing. Figure 6.8 shows the utilization of each disk at MPL = 4, 24, and 72 in the failure mode of operation. As shown in the figure, with the CD scheme, the disk utilization is roughly the same at all remaining disks. On the other hand, with the ID scheme, the workload of the failed disk is assumed only by the remaining disks in the failed ID cluster. As a consequence, the utilization of the disks in the failed ID cluster is about 14%

higher than that of all other disks. Figure 6.8 confirms the analytical result presented in Chapter 4 that the CD scheme can provide better load balancing than ID, and does not explain the performance difference in favor of ID.

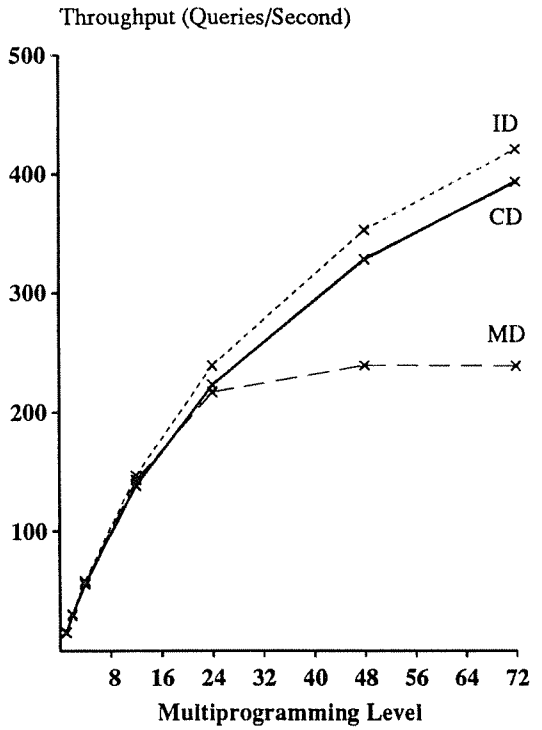

Figure 6.7: Single Tuple Selection Failure Mode

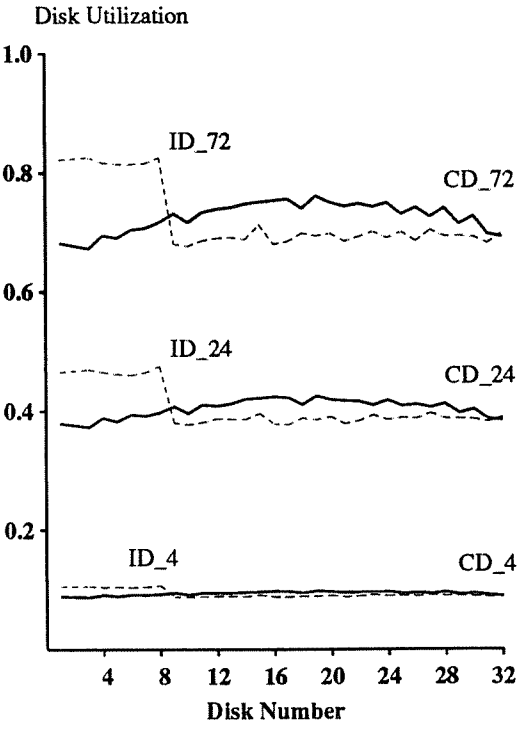

Figure 6.8: Single Tuple Selection Failure Mode

The reason for the CD scheme providing better load balancing but less throughput than ID is the following: When a disk failure occurs, a system with the CD scheme will be switched into failure mode operation; both the primary and backup fragments are used to serve disk read requests. With the ID scheme, disks in the failed cluster are also switched into failure mode operation; the corresponding backup subfragments of the failed disk (failed primary fragments) are used to serve disk read requests. Because the backup fragments with both schemes are placed on the inner half of disk tracks (as described in Section 5.3), this results in higher disk seek time in the failure mode of operation for both CD and ID schemes. In our experiments, a relation cluster contains all 32 disks whereas the size of an ID cluster is set to 8. Thus, when a disk fails, all 31 remaining disks are affected (resulting in a longer seek distance for each) with the CD scheme, whereas only 7 disks are affected with the ID scheme. Even though the CD scheme can provide better load balancing, the average disk seek time is higher than that with ID (where 24 disks remain in normal mode operation). As a result, the ID scheme provides higher throughput than CD in this particular

case (and, as we will see, only in this particular case).

When a query requires both disk reads and writes, both the primary and backup fragments are accessed even in the normal mode of operation. In such cases, the use of backup fragments to satisfy disk read requests in the event of a disk failure will have little or no impact on the disk seek time (as will be seen later, in Experiment 4). In addition, if a relation cluster is further divided into multiple chain clusters in CD, and the sizes of a chain cluster and an ID cluster are the same, the impact of a disk failure on disk seek time will be approximately the same with both schemes. To demonstrate this, we ran the same single tuple selection experiment (case B) with the model configured to have 8 nodes and one disk per node. Figures 6.9 and 6.10 show the throughput and disk service time of CD and ID with this special configuration. In these two figures, the suffixes _N and _F are used to represent the normal and failure modes of operation, respectively. As shown in these two figures, CD and ID now have comparable throughput and disk service time in both the normal and failure modes of operation.

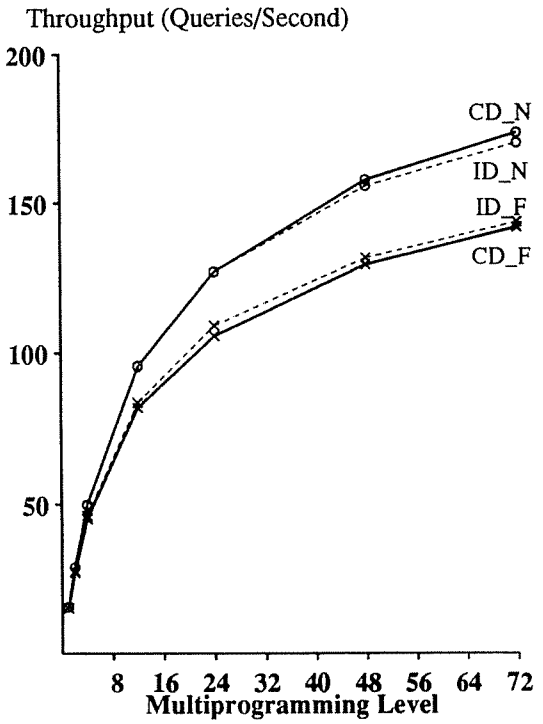

Figure 6.9: Single Tuple Selection

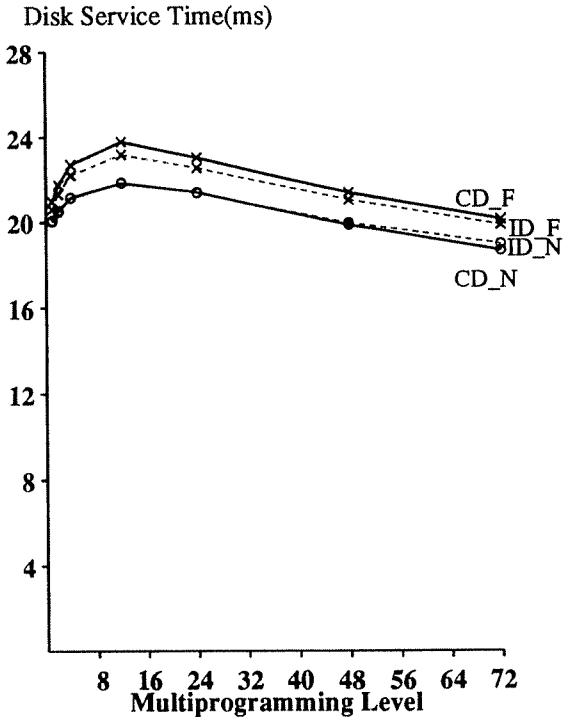

Figure 6.10: Single Tuple Selection

Experiment 2: 1% selection query using a clustered index

In this experiment, we assume that either a relation is hash partitioned or a relation is range partitioned and the clustered index is constructed on a nonpartitioning attribute. In either case, a range selection query using the clustered index (i.e. where the qualified attribute is not the range partitioning attribute) will have to be sent to all the active nodes[7] for processing in both the normal and failure modes of operation.

With the CD and ID schemes, a 1% selection query reads 2 or 3 index pages[8] and 18 data pages from each disk in the normal mode of operation. With the MD scheme, this query type reads 2 or 3 index pages and 35 data pages from each mirrored pair. With all three schemes, 1250 tuples are selected at each node and the results are returned to the user/application program. In this experiment, a node with the CD and ID schemes reads and processes two more index pages and one more data page for each query than it does with MD in the normal mode of operation. This is because, for each query, the CD and ID schemes traverse two separate indices at each node in order to read data pages from the two disks attached to the node (since that both the primary and backup fragments are distributed across both the disk drives attached to a processor), whereas each node with the MD scheme traverses only one index (B+ tree) structure. However, while the CD and ID schemes pay the overhead of reading more index and data pages, they also receive the benefit of higher intra query parallelism. As will be shown later, this increased parallelism pays off at low multiprogramming levels (before the CPU becomes the bottleneck). In addition, because a fragment in CD is half the size of that in MD, for some combinations of relation and disk page sizes (e.g. a 1M tuple relation and a 4K page size), the depth of a B+ tree is higher with MD than with CD or ID. In such cases, a node will read the same number of index pages for a 1% selection query with all three schemes.

Figure 6.11 shows the throughput of a 1% selection query. At a MPL of 1, as shown in Figure 6.11a, the CD and ID schemes provide about 60% more throughput than MD in the normal mode. This is because both disks at a node are actively serving a query with CD and ID, whereas only one disk is active at a time with the MD scheme. However, CD and ID do not provide 100% more throughput than MD at a MPL of 1 due to the fact that the CD and ID schemes read and process more pages per query than MD does (as explained in the previous paragraph). Figure

[7] In this experiment, with the ID scheme, a subquery is sent to one or two backup subfragments for processing when the corresponding primary fragment has failed. For a 1% selection query, it is very unlikely that the subquery range will intersect the qualified attribute values in two subfragments. Therefore, in most cases a subquery will be sent to a single backup subfragment for processing when the responsible primary fragment fails in ID.

[8] It reads 2 index pages in most cases. However, when the selection range matches with the qualified attribute values in more than one leaf page of an index structure, it will need to access 3 index pages to retrieve all matching tuples in the data pages.

6.11a also shows that the CD and ID schemes continue to provide higher throughput than MD until the MPL is greater than 12. When MPL > 12, the CPU is 100% utilized and becomes a bottleneck[9] with both the CD and ID schemes. Consequently, their throughput levels off after a MPL of 12. On the other hand, the throughput with MD does not level off until a MPL of 24. This is because, with the MD scheme, a node activates fewer operator processes and reads and processes fewer pages per query than it does with the CD and ID schemes. Consequently, the CPU bottleneck does not form until later. Ultimately, at a MPL of 48, the MD scheme provides about 5% higher throughput than CD and ID do. Figure 6.11a shows that there is a tradeoff between the benefit of higher intra query parallelism and the overhead of scheduling more operator processes and processing more index pages. If a system will be consistently operated under high CPU utilization (i.e., its applications are CPU bound), then the partitioning strategy/data placement algorithm used with CD and ID should be modified to have only 16 fragments[10] per relation (as in the case of MD) rather than 32 fragments per relation (as is used in this experiment). By reducing the degree of declustering with CD and ID to 16, all three schemes will activate the same number of operator processes per query and read and process the same number of pages per node. With this new arrangement, the CD and ID schemes will provide the same level of throughput as the MD scheme in the normal mode of operation.

Figure 6.11b shows the throughput of all three schemes in the event of a disk failure. At a MPL of 1, the throughput with CD and ID drops by almost 40% from the normal mode of operation when a disk failure occurs. The reason for this performance degradation is that the workload of a system with CD and ID is not evenly redistributed on a per query basis in the event of a disk failure. For each query, the portion of work originally served by the failed disk is likely to be picked up by a single disk. As a result, for each query one of the remaining disks will have to serve twice as many requests as the other disks. In addition, as described in the single tuple selection experiment, for read only queries the disk seek time is higher in the failure mode than in the normal mode with both the CD and ID schemes because in the failure mode the disk arm has to travel across all cylinders on a disk.

At a MPL of 1, a new query is not generated until the previous query has completed. Since each query takes almost twice as much time to complete as it does in the normal mode, throughput drops significantly when a disk failure occurs. At high MPLs, there are multiple queries outstanding, all generating I/O requests, and the work of

---

[9] In our simulation model (and in Gamma), the processor is responsible for transferring data from I/O channel's FIFO buffer to main memory. Without this overhead, the CPU bottleneck would form at a higher MPL.

[10] The two disks attached to a processor with CD and ID can be treated as one "logical" disk.

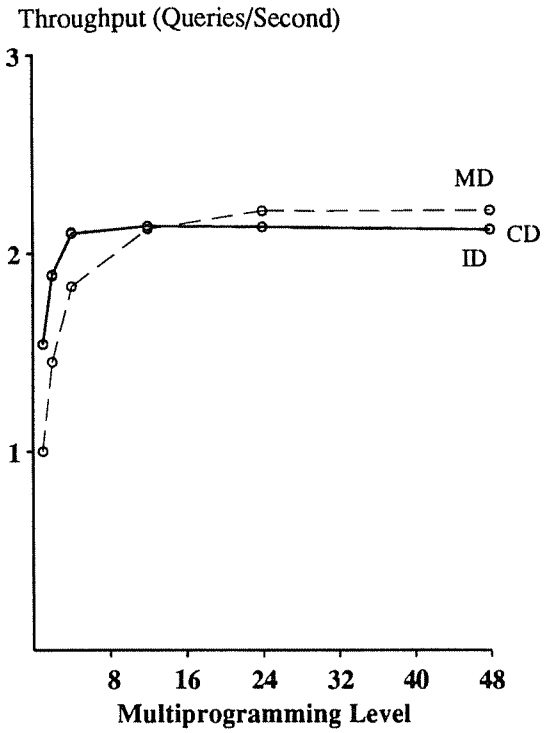

Figure 6.11a: 1% Selection
Normal Mode

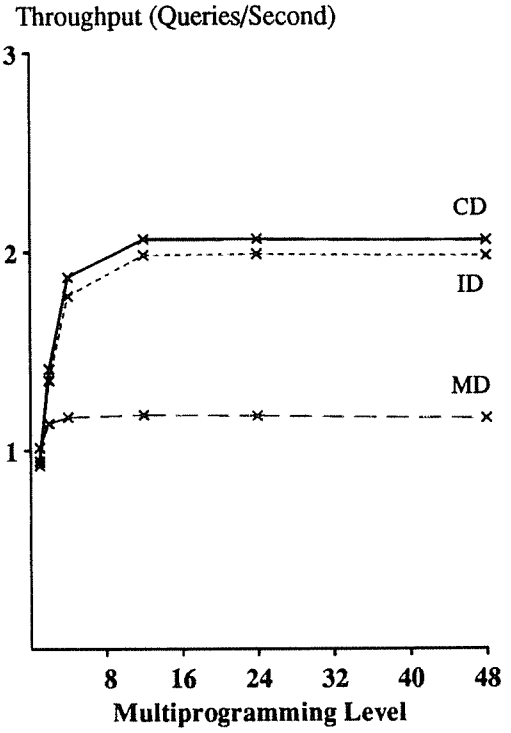

Figure 6.11b: 1% Selection
Failure Mode

the failed disk is picked up randomly by all remaining disks (since the selection range of each subquery determines which one of the remaining disks is responsible for the subquery) in the failed cluster rather than by a fixed disk. The higher the MPL is, the more likely it is that the work of the failed disk will be picked up by multiple disks in failure mode. Thus the workload in failure mode will be more evenly distributed among remaining disks at high MPLs, and the performance degradation with CD and ID will be less drastic. This is indeed shown in Figure 6.11, as when MPL $\geq$ 12, the reduction in throughput in the failure mode is about 3.5% with CD and about 8% with ID.

With the MD scheme, there is little or no performance degradation at a MPL of 1 when one disk fails. This is because there is always a disk in a mirrored pair that is idle in the normal mode for this particular query type and MPL. Thus, instead of two disks in a mirrored pair taking turns serving read requests, the remaining disk in the failed mirrored pair serves all of the requests[11]. As a result, there is little or no performance degradation with MD at

[11] Note that, when a disk completes the service of the last disk request of one query and moves its head to serve the next query, the disk seek time (distance) will be higher in the failed mirrored pair (without the benefit of shortest-seek-time-first). However, since each query generates 37 disk requests sequentially, the higher seek time between requests generated from two consecutive queries will be amortized over 37 requests and become negligible in the failed pair.

a MPL of 1 in the failure mode of operation.

Figure 6.11b shows that, at a MPL of 1, MD provides about 7% higher throughput than CD, while CD provides slightly higher throughput than ID. The reason that CD and ID provide less throughput than MD at a MPL of 1 is the following: As described above, for each query there is one disk, D, serving twice as many requests as the remaining disks with both CD and ID. This means that disk D will serve about the same number of requests as a disk in a mirrored pair for a given query. As a result, when a disk fails, the response time for a given query will be about the same with all three schemes at a MPL of 1. In addition, to obtain higher intra query parallelism in the normal mode, both the CD and ID schemes read and process more index pages and occasionally more data pages (as explained earlier). When a disk failure occurs, the load balancing algorithm has little or no effect with both CD and ID at a MPL of 1 (as explained above) and the overheads of the higher disk seek time and processing of extra pages overshadows the benefit of parallelism.

At a MPL of 2, both CD and ID begin to have a higher throughput than MD. This is because there is more overlap of the CPU and disk service times and the benefit of better load balancing and higher parallelism becomes more significant than the overheads described above. With the MD scheme, at a MPL of 2 disk #3 is more than 95% utilized while all other remaining disks are about 30% utilized. This disk workload imbalance results in a lower throughput with MD; that is, most queries are waiting in the queue of disk #3 while all other disks have no requests to serve. At a MPL of 4, the remaining disk in the failed mirrored pair is fully utilized and truly becomes a bottleneck. Consequently, the MD throughput levels off when MPL $\geq$ 4. With the CD and ID schemes, no disk bottleneck forms and the throughput continues to increase until a MPL of 12. At that point, the CPU is fully utilized and becomes the bottleneck.

When MPL $\geq$ 12, the throughput with both CD and ID levels off and is, respectively, about 80% and 70% higher than it is with MD. Figure 6.11b also illustrates that CD provides higher throughput than ID throughout the entire experiment in the failure mode of operation. This is because the CD scheme provides better load balancing while both schemes incur about the same level of overhead in the event of a disk (node) failure.

Figures 6.12 and 6.13 show, respectively, the utilization of each disk at MPL = 1 and 48 and of each processor at MPL = 1, 12 and 48 in the normal mode of operation. As illustrated by these two figures, a 1% selection query using the clustered index is CPU bound. It is also clear that a system with CD or ID has much higher CPU and disk utilization than a system with MD at a MPL of 1 because the processors and disks with both CD and ID

serve more requests than they do with MD. At a MPL of 48, on the other hand, the disk utilization with MD is higher than it is with CD and ID. This is because the CPU is 100% utilized and has become a bottleneck at a MPL of 12 with CD and ID, whereas it is not fully (100%) utilized until a MPL of 24 with MD (as shown in Figure 6.13). Therefore, a system with MD provides somewhat higher throughput (thus processing more disk requests) than either CD or ID at a MPL of 48.

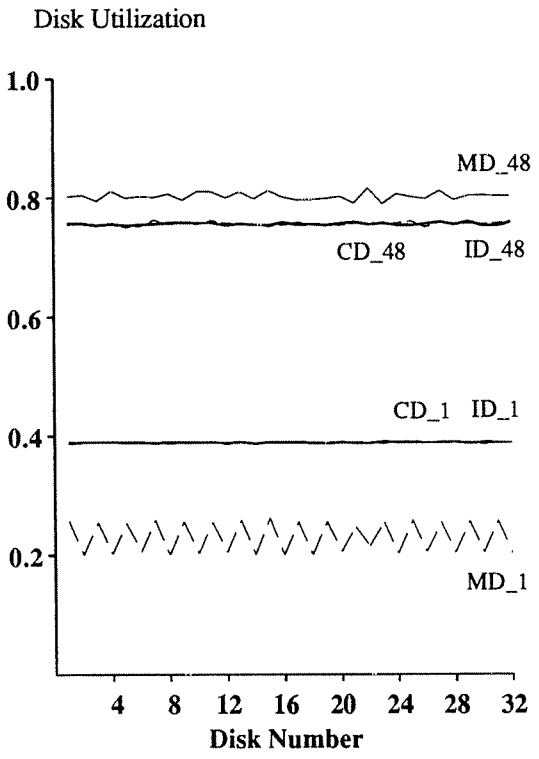

Figure 6.12: 1% Selection, No Failure

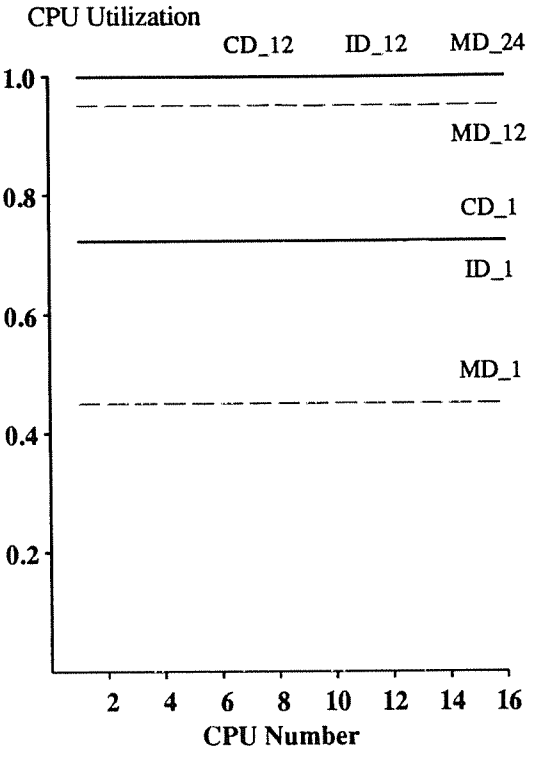

Figure 6.13: 1% Selection, No Failure

Experiment 3: 0.1% selection query using a nonclustered index

In this experiment, we assume that the qualified attribute is different from the partitioning attribute (or a relation is hash partitioned). In this case, a range selection query using a nonclustered index will have to be sent to all active nodes for processing in both the normal and failure modes of operation. In the normal mode of operation, a 0.1% selection query reads 2 or 3 index pages and 63 data pages from each disk with the CD and ID schemes whereas it reads 2 or 3 index pages and 125 data pages from each mirrored pair of disks with the MD scheme. With all three schemes, 125 tuples are selected at each node and the results are returned to the user/application program. As in Experiment 2, the 0.1% selection query in this experiment reads and processes more index and data pages

with both the CD and ID schemes than with the MD scheme. By paying this overhead, however, both the CD and ID schemes receive the benefit of higher intra query parallelism.

Figure 6.14 shows the throughput of this query. As shown in Figure 6.14a, at a MPL of 1, CD and ID provide about 90% more throughput than MD does in the normal mode of operation. As explained in Experiment 2, the major reason for the big difference in throughput is that both disks at a node are simultaneously serving a query with the CD and ID schemes, whereas only one disk is active at a time with the MD scheme. In this experiment, the throughput difference between MD and the other two schemes at a MPL of 1 is higher than it is in Experiment 2 and the difference is close to 100%. The reason for the bigger throughput difference in this experiment is as follows: First, the query type in this experiment is I/O bound, whereas it is CPU bound in Experiment 2. Doubling the number of disks serving a query will thus cut the response time by one half (doubling the throughput) here. With CPU bound applications, however, the response time of a query is affected mainly by the CPU utilization and the degree of overlap between CPU execution and disk execution. Second, a node reads and processes about 128 pages for each query in this experiment, whereas it reads and processes only 37 pages per query in Experiment 2. The reading and processing of two extra index pages and one extra data page per query account for about 2% of the overhead in this experiment, whereas they account for more than 8% of the extra CPU overhead in Experiment 2.

As indicated in Figure 6.14a, both CD and ID provide higher throughput than MD throughout the entire experiment in the normal mode of operation. At a low MPL, the higher parallelism of CD and ID is the major reason that they perform better than MD. At a high MPL (e.g., MPL $\geq$ 12), all disks are likely to be active. In such regions, the throughput difference between MD and the other two schemes is due partly to the different disk scheduling algorithms used in the different schemes and partly to the difference in the degree of query parallelism. As indicated in Figure 6.14a, both CD and ID provide significantly (about 39%) higher throughput than MD at a MPL of 48.

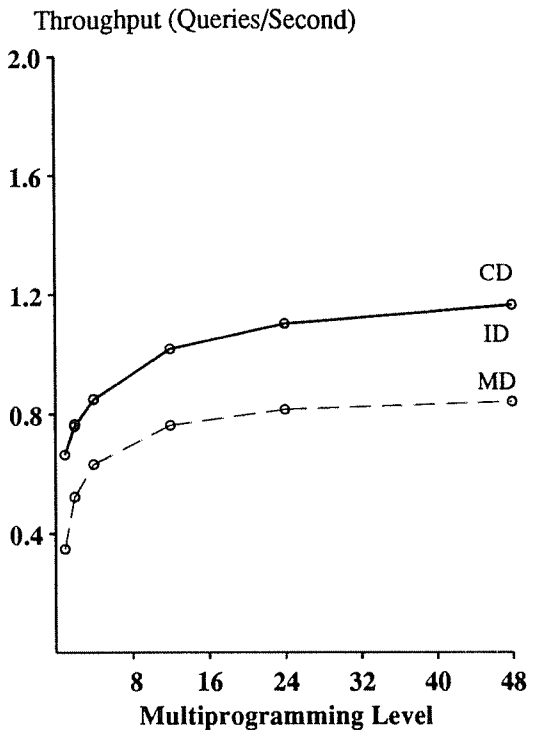

Figure 6.14a: 0.1% Selection
Normal Mode

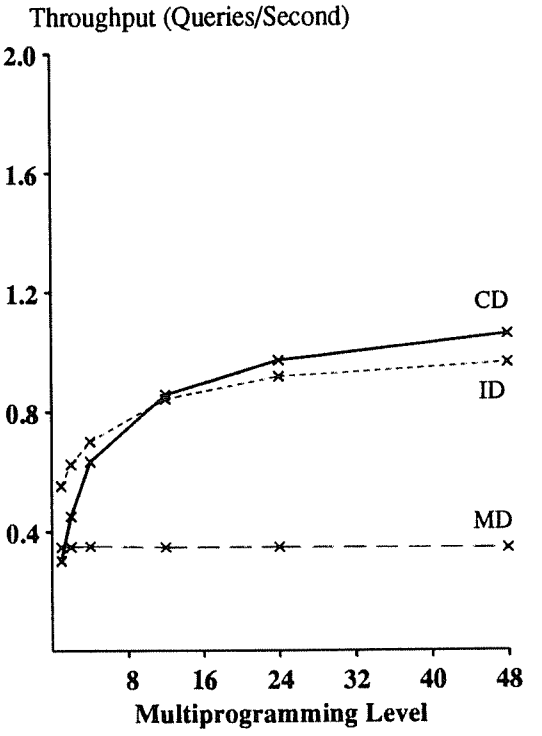

Figure 6.14b: 0.1% Selection
Failure Mode

Figure 6.14b shows the throughput of all three schemes in the event of a disk failure. At a MPL of 1, MD provides slightly higher throughput than CD, while ID provides significantly higher throughput than CD there. The reason for this difference is that with CD (as explained in Experiment 2) the throughput at a MPL of 1 decreases significantly due to the uneven distribution of the workload. When a disk failure occurs, a subquery with 0.1% selectivity is very unlikely to intersect with both responsible ranges of a given fragment. Thus, for a given query, two of its subqueries are likely to be served by one disk, leading to a 100% workload increase at this disk. Since a query does not complete until all nodes serving the query complete their service, the query response time increases by 100% and the throughput decreases correspondingly by about 50% at a MPL of 1.

With the ID scheme, the workload of the failed disk is distributed evenly[12] among the remaining disks in the failed ID cluster. As a result, at a MPL of 1 the increase in workload on disks in the failed ID cluster is only 14% and the decrease in throughput with ID is less drastic than with CD. The increased workload together with the higher disk seek time in the failed ID cluster results in about a 16% drop in throughput at a MPL of 1. With the MD

[12] It is programmed that way in the simulation model.

scheme, there is no performance degradation at a MPL of 1 when a disk failure occurs. This is because only one out of two disks in a mirrored pair is active at a time in the normal mode of operation (as explained in Experiment 2). The failure of a disk thus does not affect the response time or throughput in any noticeable way.

At a MPL of 2, one of the MD disks becomes a bottleneck and the throughput levels off. On the contrary, the throughput with both CD and ID continues to increase as the MPL is pushed higher, mostly because of better load balancing. Figure 6.14b also shows that ID continues to provide higher throughput than the CD scheme up to a MPL of 4. When MPL > 4, however, the benefit of the better load balancing of CD begins to dominate. As indicated in Figure 6.14b, CD starts to provide even higher throughput than ID when the MPL is greater than 12. At a MPL of 48, the throughput with CD is about 10% and 200% higher than with ID and MD, respectively.

Figure 6.15 shows the disk service time of all three schemes in the normal mode of operation. As shown in the figure, the disk service time with MD is about 18.9 ms and is noticeably (1.2 ms) higher than it is with CD and ID at a MPL of 1. The reason is the fragment size difference between MD and the other two schemes, as explained in Experiment 1. With a nonclustered index access method, a disk arm will seek randomly within a fragment to serve requests generated by a query at a MPL of 1. Since MD has a larger fragment size, it will also have a longer average seek time. At a MPL of 2, however, the disk service time with CD and ID increases to about 22 ms, which is about a 4.5 ms increase over that at a MPL of 1. The big jump in disk service time from a MPL of 1 to 2 with CD and ID is due mainly to disk head contention between requests generated from two different queries. Because different queries are likely to access different relations (fragments), a disk arm will seek back and forth between two fragments and will thus have higher disk seek times. With the MD scheme, the disk head contention is less severe because only one disk in a mirrored pair is active at a time for a query. At a MPL of 2, if requests from two different queries were always directed to two separate disks, there would still be no disk head contention. Because of the random nature of the queries, however, requests from two distinct queries will occasionally be directed to the same disk and the disk head contention will occur. Because the disk head contention is less frequent with MD than it is with CD and ID, the increase in disk service time is smaller with MD than it is with CD and ID. As shown in Figure 6.15, the disk service time with CD and ID is about 2 ms higher than it is with MD at a MPL of 2. This trend continues until a MPL of 4. When the MPL is greater than 4, the elevator algorithm used in CD and ID then kicks in, and the disk service time starts to decline, whereas the disk service time with MD is essentially unchanged. Consequently, the average disk service time with MD is about 6 ms (34%) higher than it is with both CD and ID at a

MPL of 48.

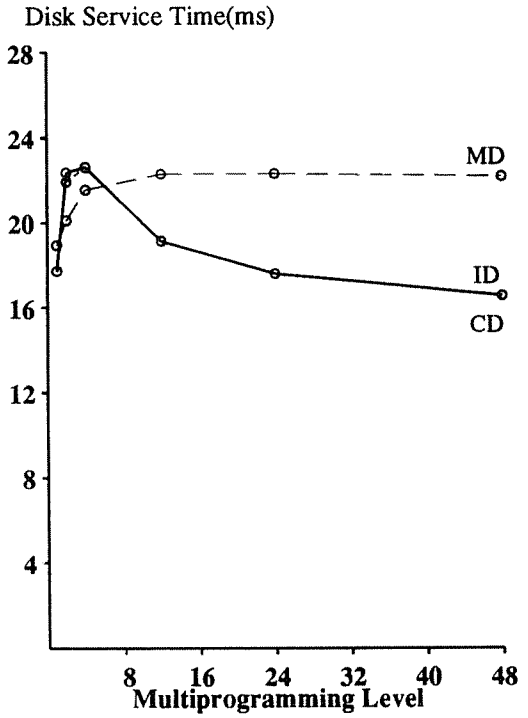

Figure 6.15: 0.1% Selection, No Failure

Table 6.3 shows the percentage difference in throughput and disk service time between MD and the other two schemes in the normal mode of operation. The second column shows how much more throughput the CD and ID schemes provide than the MD scheme, while the third column shows how much more time a disk with the MD scheme takes to serve a disk request than it does with CD or ID. (In the third column, a negative sign is used to indicate that a disk with MD takes less time to serve a request than it does with CD or ID.) As indicated in the table, at one end of the spectrum (MPL = 1), the throughput difference results mainly from the difference in the degree of intra query parallelism. At the other end of the spectrum (MPL = 48), the throughput difference results mainly from the difference in disk service times due to the different disk scheduling disciplines.

MPL	Percentage Difference In Throughput	Disk Service Time
1	90.0%	+ 7%
2	45.6%	-10%
4	34.5%	- 5%
12	33.6%	+21%
24	35.3%	+27%
48	38.5%	+34%

Table 6.3

Figures 6.16 and 6.17 show the utilization of each disk and each processor at MPL = 1 and 4 in the failure mode of operation[13]. As shown in Figure 6.16, the CD scheme provides much better load balancing in distributing the workload among the remaining operating disks than ID and MD do. At a MPL of 4, the disk in the failed mirrored pair and disks in the failed ID cluster are 100% utilized and become bottlenecks. On the other hand, the utilization of all operating disks with CD is about 80% at a MPL of 4.

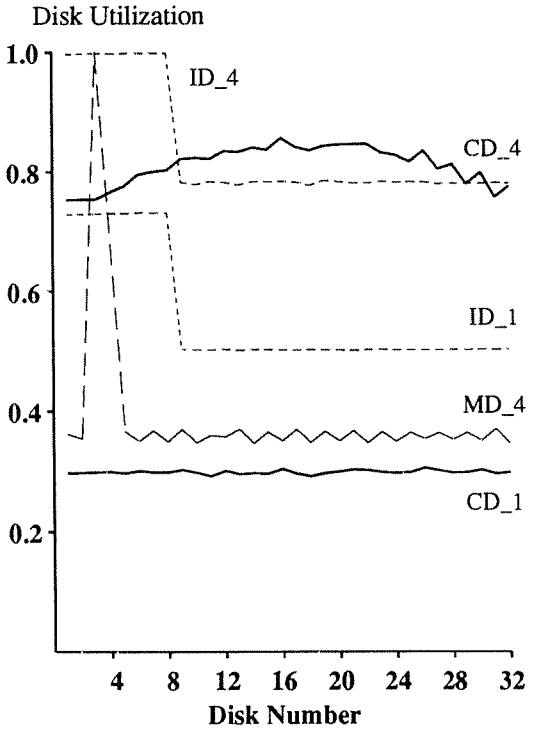

Figure 6.16: 0.1% Selection Failure Mode

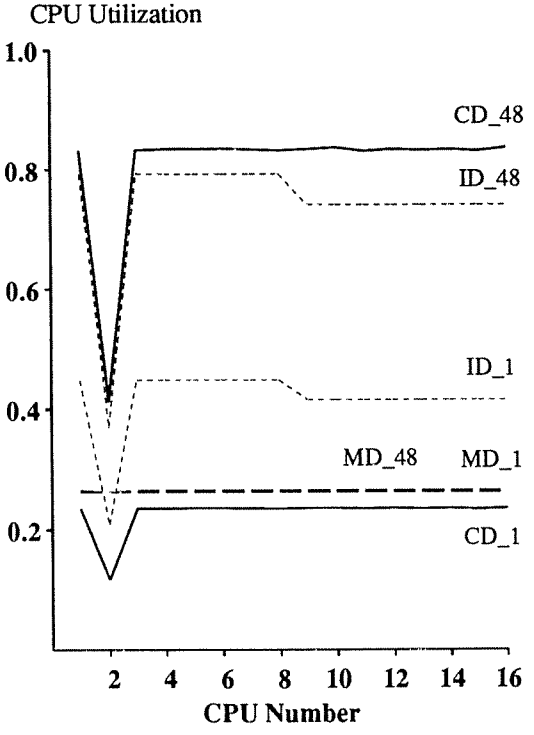

Figure 6.17: 0.1% Selection Failure Mode

[13] Disk utilization with MD at a MPL of 1 is similar to that at MPL of 4. It is not shown to make the figure more readable.

Figure 6.17 shows that the utilization of processors with MD is about the same at MPLs of 1 and 48. The reason is that a system with MD processes the same number of queries at MPLs of 1 and 48 when a disk failure occurs. This was shown in Figure 6.14b; the throughput in the failure mode is essentially the same throughout the entire range of MPLs with the MD scheme since the disk in the failed mirrored pair is fully utilized and becomes a bottleneck at a MPL of 2. Figure 6.17 also shows that CD provides better load balancing at the processor level than ID does, as would be expected.

6.4.2. Update Queries (Experiments 4 - 6)

For an update query, both the primary and backup copies of a data item are updated. With chained declustering and interleaved declustering, the backup copy is stored on a disk attached to a different processor (node) from the one storing the primary copy. A major overhead associated with updating a remote backup copy in either scheme is sending updated pages through the communication network to the node storing the backup copy. This includes extra CPU cycles for packaging, sending, and receiving network packets, as well as a wire delay in the communication network. On the other hand, with mirrored disks the backup copy is stored on the same processor as the primary copy, and handled at the controller level, so no extra CPU cycles are required to update the backup copy. However, MD incurs a different kind of overhead when updating data items. With the mirrored disk strategy, a CPU initiates a single disk write operation for each updated page; the disk controller then schedules the disk write request and writes the page to both the primary and backup copies simultaneously. This means that the disk seek distance for a write operation in MD is the maximum of two disk seeks. Bitton and Gray [Bitt88] showed that the average seek distance for write operations with mirrored disks is 0.47n, where n is the total tracks of a disk, which is 0.14n higher than the average seek distance of a single disk.

Three query types are studied in this section: a single tuple update query using a clustered index, a 1% update query using a clustered index, and a query that selects 1% of the tuples and then updates between 10% and 50% of the selected tuples. We again assume that relations are hash partitioned in the simulation experiments. A single tuple update query will thus be sent to a single node for processing, while the other two queries will be sent to all nodes. Upon receiving an update query, a node retrieves the appropriate data page(s) through the clustered index and updates the tuple(s) on each page before storing the updated page(s) back on disk. In addition, with CD and ID, an updated page is sent through the communication network to its backup copy node in order to update the backup copy. In our experiments, an update transaction is not committed until both the primary and backup copies have

been updated. Finally, the updated tuple attribute is assumed to be different from the indexed or partitioning attribute, so no data movement or index structure reorganization is required after the execution of the query.

Experiment 4: Single tuple update query

This query type reads a data page through a clustered/nonclustered index, updates a tuple within the page, and writes the data page back to disk (updating both the primary and backup copies). This query type is similar to the account balance update of a debit-credit (TP1) transaction[14] [Anon85].

Similar to Experiment 1, with the ID scheme, a single tuple update query will sometimes need to be sent to multiple backup subfragments for processing in the event of a node failure. When the relation being updated is range partitioned and the clustered index is also constructed on the partitioning attribute, a single tuple update query can be directed to a single backup subfragment in the event of a node failure. With all other combinations of partitioning strategies and indexing schemes, however, this query must be directed to multiple backup subfragments for processing in the failure mode of operation.

Case A: In this experiment, we assume that the relation being updated is hash partitioned and an index is constructed on the partitioning attribute. Figure 6.18 shows the throughput of a single tuple update query. As shown in Figure 6.18a, all three schemes provide comparable performance at low multiprogramming levels. Upon close examination, however, one can see that MD actually provides slightly higher throughput than CD when the MPL <= 12. This is because, at a low MPL, the overhead of updating remote backup copies with CD or ID is higher than that with MD. As the MPL is pushed higher, though, the overhead of synchronizing disk writes in the mirrored disk scheme also becomes higher. This is because two disk arms in a mirrored pair are more likely to be farther apart when more than one read request is served by a mirrored pair. On the other hand, the overhead of a remote update with CD and ID is a function of the page size and is thus not affected by a higher MPL. When the MPL is higher than 24, the overhead of synchronizing disk writes becomes higher than the overhead of remote updates. As a result, the CD and ID schemes provide significantly higher throughput than the MD scheme after MPL > 24 in the normal mode of operation. Figure 6.18a also shows that CD provides slightly higher throughput than ID. The reason for the difference is the following: A backup fragment in ID is broken into multiple subfragments and each sub-

[14] As part of a TP1 transaction, a customer account record is retrieved through the account number and the account balance is updated.

fragment has its own index structure. As a result, a system with the ID scheme stores slightly more data/index pages for a backup fragment on a disk than it does with the CD scheme. Consequently, ID has a slightly longer seek time than CD.

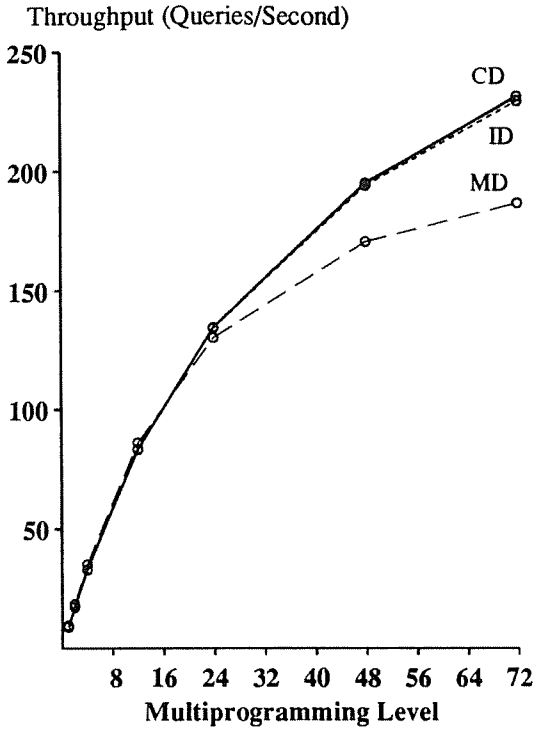

Figure 6.18a: Single Tuple Update
Normal Mode

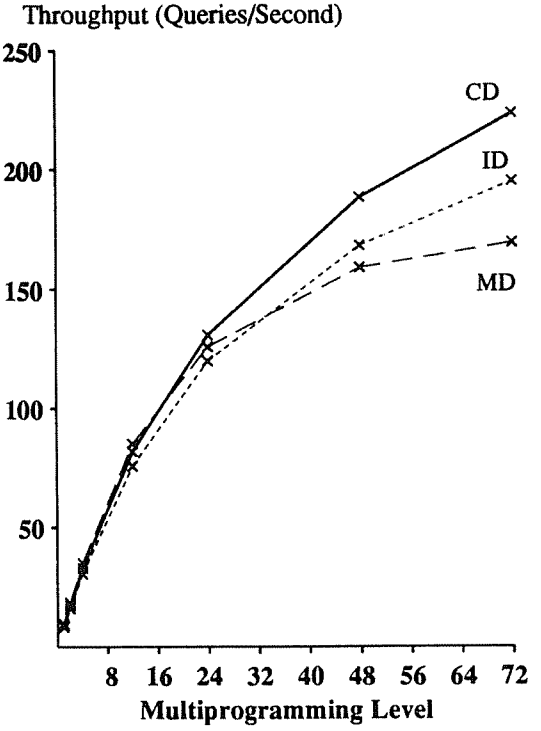

Figure 6.18b: Single Tuple Update
Failure Mode

In the event of a disk failure (Figure 6.18b), the performance of CD degrades only slightly while the degradation in performance with ID and MD is much more significant. At a MPL of 72, for example, Figures 6.18a and 6.18b show that the throughput drops by 3.3% with the CD strategy, by 14.9% with the ID strategy, and by 9.2% with the MD strategy when a disk failure occurs. The reason for the small degradation in throughput for CD is because the CD scheme is able to evenly redistribute the workload among the remaining 31 disks. With the ID scheme, the workload of the failed disk is distributed only among the remaining disks in the failed ID cluster. This results in a higher workload for the disks in the failed cluster. In addition, a query originally served by the failed disk (primary fragment) is sent to multiple backup subfragments for processing[15], resulting in the overhead of read-

[15] Only one subfragment finds and updates the matching tuple, however.

ing and processing extra index pages. The load imbalance plus the overhead of reading and processing extra index pages together result in a greater performance degradation with ID than with CD in the event of a disk failure.

With the MD scheme, the remaining operational disk in the failed mirrored pair must assume the entire workload of the pair. However, unlike in the case of single tuple selection, the decrease in throughput of the MD scheme in the event of a disk failure is much less than 50%. There are two major reasons for this behavior. First, with mirrored disks, write operations are directed to both disks in a mirrored pair. Because an update query issues both read and write disk requests, when a disk failure occurs, the remaining disk in the failed pair will have to pick up only the read requests originally served by the failed disk. This means that the increase in the number of disk requests in the remaining disk will be less than 100%. In particular, in this experiment each query reads 3 pages and writes one page. Given two such queries, each disk in a mirrored pair is responsible for 3 disk reads and 2 disk writes (assuming that the workload is uniformly distributed between the two disks). When a disk failure occurs, the remaining disk in the failed pair will be responsible for 6 disk reads and 2 disk writes, resulting in a 60% increase in the number of disk requests. With a 20% buffer hit ratio for read requests (the number used in our experiments), the increase in disk requests further decreases to roughly 54%.

Second, with the mirrored disk scheme, the disk service time for a write request is the maximum service time of the two disks serving the request [Bitt88]. Disk service time is composed of disk seek time, arm settle time, rotational latency, and data transfer time. For a pair of two disks serving the same request, the arm settle time and data transfer time are the same. However, both the seek time and rotational latency[16] of the two disks will differ and will depend on the previous locations of the disk heads. Assuming random requests, Bitton and Gray [Bitt88] showed that the expected seek distance for synchronized disk writes is 0.46n (n is the disk data band) which is 0.13n greater than the expected seek distance of 0.33n for a single disk write. For the type of disks used in the simulation model, this translates to about 2.8 ms of overhead for each write operation. If the disk spindles are not synchronized, the expected rotational latency for synchronizing N disks is N/(N+1)*16.667 ms [Chen90]. For N = 2, the expected rotational latency is 11.11 ms, which is 2.8 ms longer than the expected rotational latency for a single disk. In a failed mirrored pair, however, there is no need to synchronize the two disk arms. Without the overhead for synchronizing the disk arms, the remaining disk in a failed mirrored pair can process write requests much more

[16] Unless the spindles of the two disks are synchronized.

efficiently (with less service time) than a normal mirrored pair. For a failed mirrored pair, the shortened disk service time for write operations will offset some of the impact of the increase in the number of disk requests. Consequently, the impact of a disk failure on system performance is less significant than in the case of read only queries.

When the MPL is less than 24, there is little or no performance degradation with both the CD and MD strategies in the failure mode of operation. The reason is similar to that of the selection query case described in Section 6.4.1 (Experiment 1). That is, at low MPLs, both the CPU and disk are under utilized in the normal mode of operation. The load increase on a disk due to failure of some other disks will increase its utilization but have little impact on throughput. On the contrary, the performance degradation with the ID scheme is more significant. This is because the workload increase is higher with ID due to the fact that more index pages are read and processed in the failure mode. As the MPL is pushed higher (MPL $\geq$ 24), the degradation in throughput also becomes higher with all three schemes.

The update query used in this experiment is I/O bound. If an update query is CPU bound, the MD scheme may provide better performance than CD in the normal mode of operation. This is because, with the CD scheme, extra CPU cycles are needed to send update information to the remote processor that stores the backup copy for an update query. In addition, the remote processor will consume extra CPU cycles in order to receive the update information and to initiate a disk write operation to update the backup copy. Because advances in CPU technology have occurred much faster than those of disk technology [Joy85, Fran87], we believe that most database applications are/will be likely to be I/O bound.

Figure 6.19 shows the disk service time in the normal mode of operation. As indicated in Figure 6.19, the disk service time with the MD scheme is higher than with the other two schemes. At low MPLs, the difference mainly results from the overhead of synchronizing disk writes in the MD scheme. As shown in Figure 6.19, at a MPL of 1, the average disk service time with MD is 25.1 ms, whereas it is 21.6 ms with CD and 21.7 ms with ID. At high MPLs, the difference in disk scheduling algorithms results in a greater difference in disk service time between MD and CD. At a MPL of 72, the average disk service time is 30.9 ms and 24 ms with MD and CD, respectively.

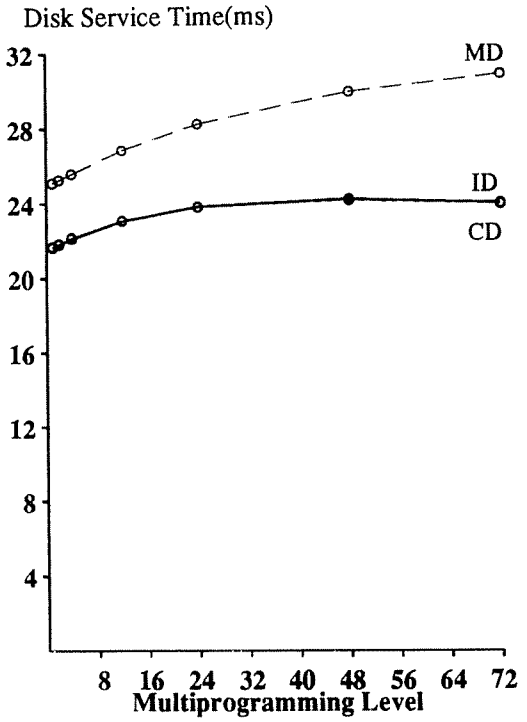

Figure 6.19: Single Tuple Update, No Failure

Figure 6.20 illustrates the utilization of each disk for the single tuple update query. As in the single tuple selection case, with the CD strategy, the disk utilization is roughly the same at all disks in both the normal and failure modes of operation. On the contrary, both the ID and MD schemes again show a highly uneven distribution of the workload among their active disks in the event of a failure. Unlike in the read only case (Figure 6.5), however, Figure 6.20 shows that with the MD strategy the utilization of disk #3 is only about 50% higher (compared to 100% in Experiment 1) than the remaining disks at a MPL of 4. Also, the disk bottleneck does not fully form until a MPL of 72 with the MD scheme. Figure 6.20 also shows that the workload within a mirrored pair is more evenly distributed than in the selection case (see Figure 6.5). This is because disk writes in a mirrored pair are served by two disks synchronously (i.e., two arms will move to the same cylinder after a write), so read requests following disk writes will be served by two disks alternatively. As a result, disk requests for update queries are more evenly distributed between two disks in a mirrored pair than for read only queries.

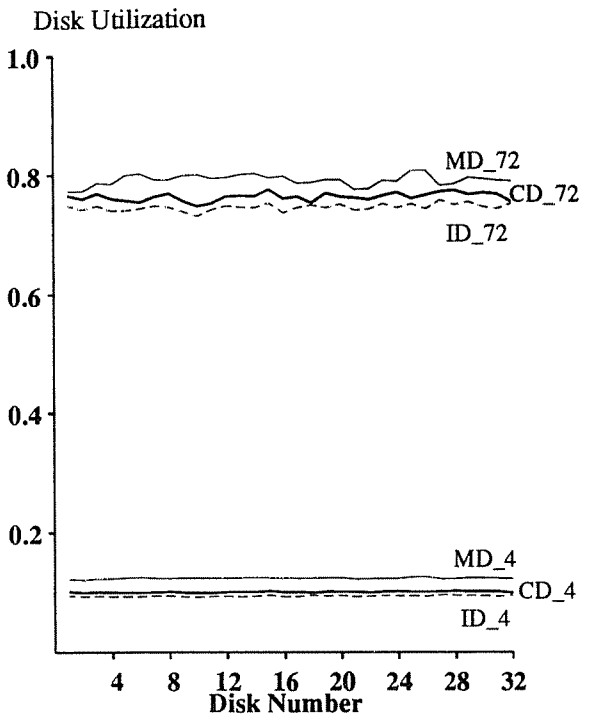

Figure 6.20a: Single Tuple Update
Normal Mode

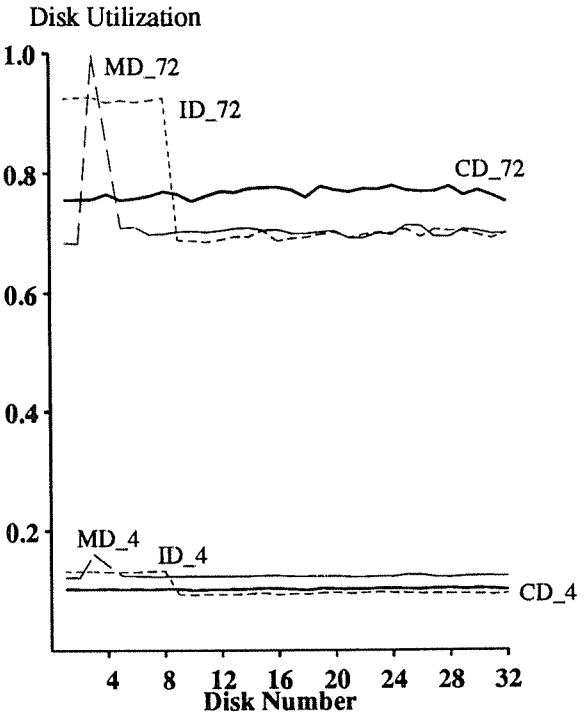

Figure 6.20b: Single Tuple Update
Failure Mode

Case B: In this experiment, we assume that the relation being updated is range partitioned AND a clustered index is also constructed on the partitioning attribute. With this combination of partitioning strategy and indexing scheme, the selection part (which is a selection on the partitioning attribute) of a single tuple update query will be directed to a single node for processing with all three schemes in both the normal and failure modes of operation. With the CD and MD schemes, the results are identical to those of case A above. With the ID scheme, the results in the normal mode of operation are also the same as they are in case A, but in the failure mode the results are significantly different. In this case there is neither CPU processing overhead for activating additional operator processes nor disk I/O overhead for accessing extra index pages with the ID scheme .

Since the results in the normal mode of operation are identical to that of case A above, in the following we will only show the results in the failure mode of operation. Figure 6.21 shows the average throughput of a single tuple selection query using a clustered index for the CD and ID schemes (the results of MD are also shown in the figure as a reference). As shown in Figure 6.21, both the CD and ID schemes provide approximately the same throughput throughout the entire experiment. The ID scheme provides slightly higher throughput than CD does due to the fact that all remaining 31 disks participate in the failure mode operation with CD, whereas only 7 disks

operate in the failure mode with ID. As explained in Experiment 1, the disk service time in the failure mode is higher than it is in the normal mode. However, the difference in disk service time between the normal and failure modes is much smaller with an update query; the reason is that a disk is already accessing both the primary and backup fragments (since an update writes to both fragments) in the normal mode of operation. The use of backup fragments to satisfy read requests in the failure mode thus has much less of an impact on disk service time. As a result, ID provides only slightly higher throughput than CD does in the failure mode of operation even though there are many more disks operating in the failure mode with CD.

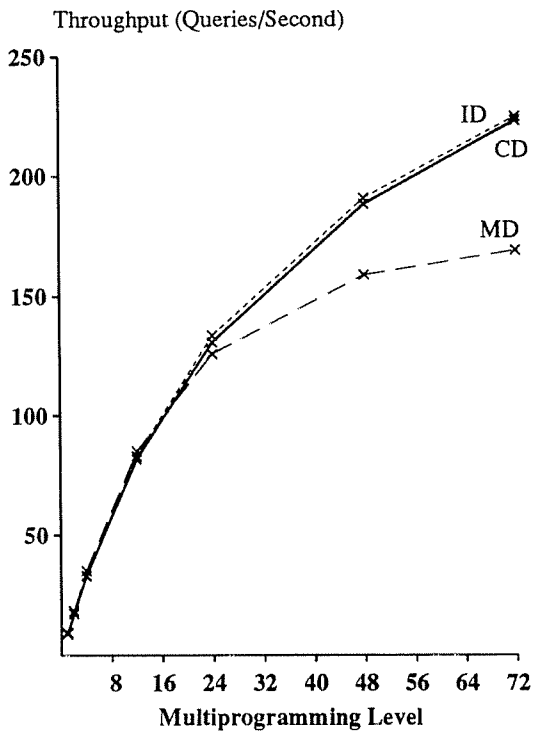

Figure 6.21: Single Tuple Update Failure Mode

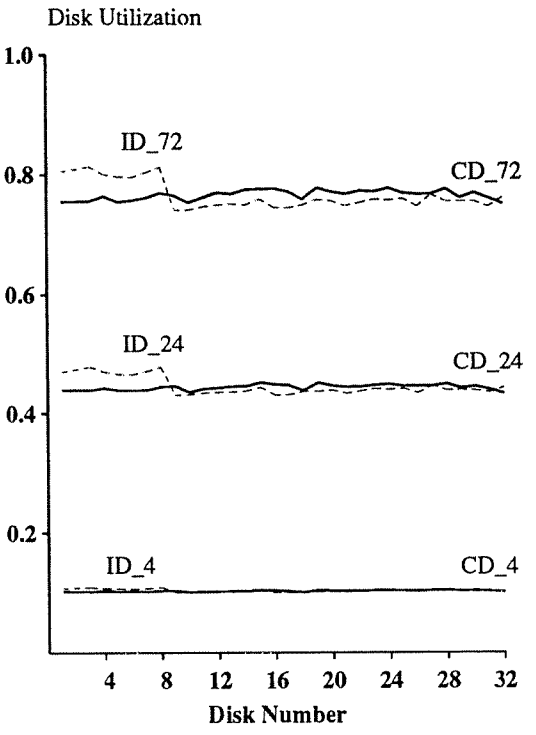

Figure 6.22: Single Tuple Update Failure Mode

Figure 6.22 shows the utilization of each disk at MPL = 4, 24, and 72 in the failure mode of operation. Again, the disk utilization is roughly the same at all remaining disks with the CD scheme. With the ID scheme, however, the workload of the disks in the failed clustered is higher than that of the remaining disks.

As explained in Experiment 1, if a relation cluster is further divided into multiple chain clusters and the sizes of a chain cluster and an ID cluster are set to be the same, the impact of a disk failure on disk seek time will be approximately the same with both schemes. To demonstrate this for update queries, we ran a set of experiments

with the model configured to have 8 nodes and one disk per node. Figures 6.23 and 6.24 show the throughput and disk service time of CD and ID with this special configuration. Again, the suffixes _N and _F are used to represent respectively the normal and failure modes of operation. As shown in Figure 6.23, CD actually provides slightly higher throughput than ID does in both the normal and failure modes of operation. Figure 6.24 shows that there is little difference in disk service times between CD and ID in both the normal and failure modes of operation.

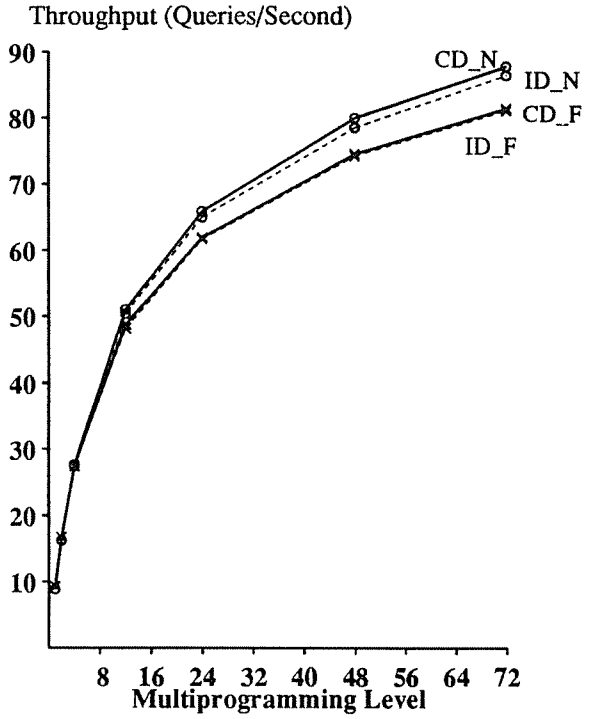

Figure 6.23: Single Tuple Update

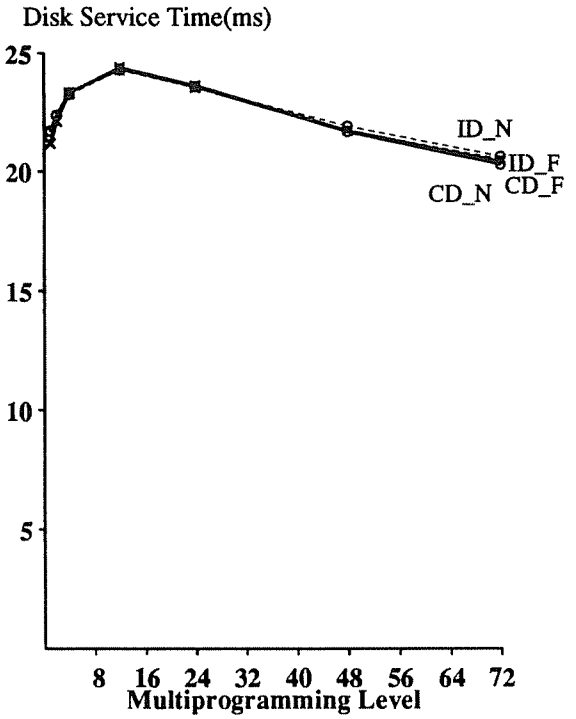

Figure 6.24: Single Tuple Update

Experiment 5: 1% update query using a clustered index

As in Experiment 2, in this experiment we again assume that either a relation is hash partitioned or a relation is range partitioned and the clustered index is constructed on a nonpartitioning attribute. In either case, a 1% update query using the clustered index will have to be sent to all active nodes for processing in both the normal and failure modes of operation. As in Experiment 2, with the ID scheme a subquery may be sent to one or two backup subfragments for processing when the corresponding primary fragment has failed.

The processing of a 1% update query is similar to that of a 1% selection query described in Experiment 2. The major difference is that, instead of returning matching tuples back to the user, a 1% update query updates all the

matching tuples. With the CD and ID schemes, a 1% update query reads 2 index pages and 18 data pages from each disk, updates all data pages read, and writes them back to the disks storing either the primary or backup fragment. With the MD scheme, a 1% update query reads 2 index pages and sequentially updates 35 data pages from each mirrored pair. With each scheme, 1250 tuples are selected and updated at each node. As explained in the 1% selection query case, two more index pages and one more data page at each node are read and processed with the CD and ID schemes than with the MD scheme.

Figure 6.25 shows the throughput of a 1% update query. As shown in Figure 6.25a, at a MPL of 1, the MD scheme provides about 25% more throughput than CD or ID in the normal mode of operation. There are two main reasons for the difference in throughput at a MPL of 1. First, in CD and ID, read and write requests from two subqueries of the same query will compete for disk service, resulting in a higher disk service time; instead of reading and updating 1% of the data pages sequentially, the disk heads seek between a primary fragment, where pages are read and written on behalf of one subquery, and a backup fragment, where pages are written on behalf of another (remote) subquery. In contrast, disk pages are sequentially read and written with the MD scheme. Second, CD and ID read and process more pages per query than MD does as explained in the previous paragraph. At a MPL of 1, the above two overheads with CD and ID are significantly higher than the benefit of higher intra query parallelism.

When the MPL is increased from 1 to 2, the throughput with the MD scheme drops by 20%. This is because requests from different queries begin to compete for disk service. Consequently, the disk arms will move randomly among pages (cylinders) requested by different queries, resulting in a higher disk service time. Throughput continues to decrease slightly from MPLs of 2 to 4 with MD because of increased disk contention. When the MPL is greater than 4, one or more disks are fully utilized and become a bottleneck with the MD scheme. As a result, the throughput remains unchanged after MPL > 4.

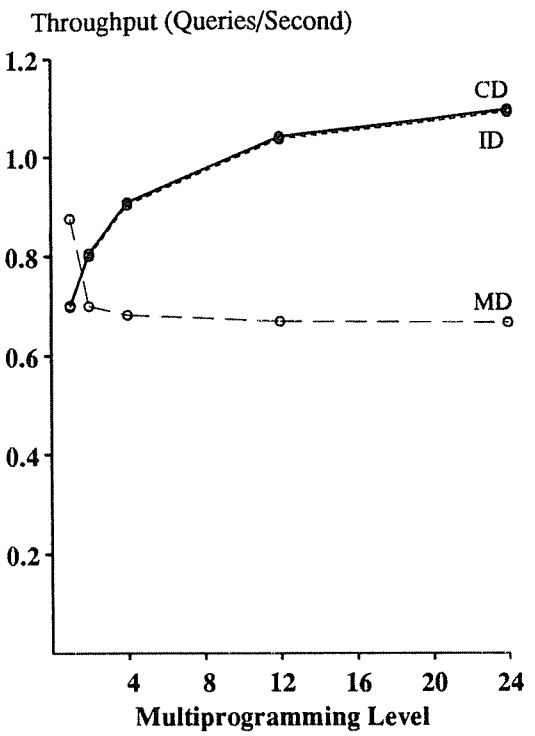

Figure 6.25a: 1% Update
Normal Mode

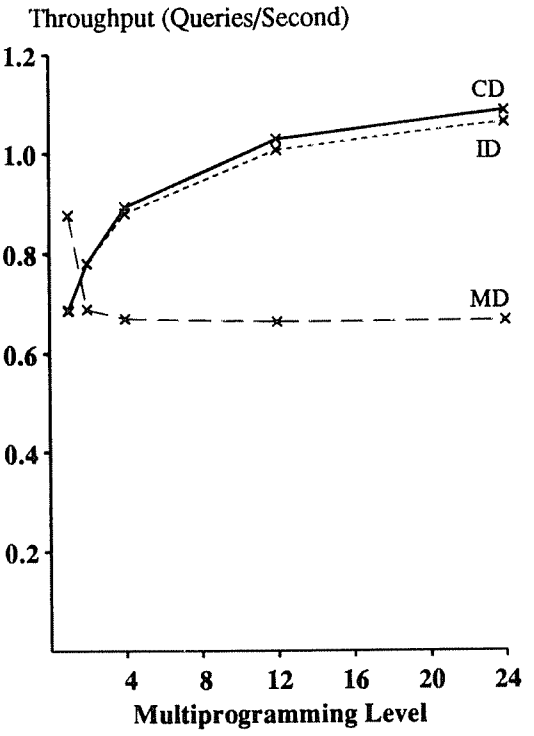

Figure 6.25b: 1% Update
Failure Mode

With both the CD and ID schemes, throughput increases by about 14% when the MPL is increased from 1 to 2. Although the disk service time with both CD and ID also increases (only slightly) when the MPL is increased from 1 to 2, the service time savings from higher inter query parallelism (more overlap between the CPU and disk operations) more than offsets the increase in disk service time. At a MPL of 1, there is concurrent execution between the disks and the CPU at a node (due to select and store processes), but the execution time is not completely overlapped. When the MPL is pushed higher, the chance for overlapping disk and CPU operation at a node is increased. The throughput with CD and ID continues to increase after a MPL of 2 because of this effect plus the fact that the disk service time with CD and ID starts to decrease when the MPL is greater than 2. At a MPL of 24, Figure 6.25a shows that the CD scheme provides slightly higher throughput than ID and significantly higher (65% higher) throughput than MD. The big throughput difference between CD and MD results from the differences in disk service time and in the degree of intra query parallelism between these two schemes. As will be seen later, the average disk service time per request with MD is about 55% higher than that with CD.

In the event of a disk failure (Figure 6.25b), there is only a small performance degradation with all three schemes. With the CD scheme, there is only a small workload increase on each of the remaining disks because the

workload of the failed disk is evenly distributed among them. With the ID scheme, the workload increase on the disks in the failed ID cluster is also small, but the increase is higher than it is with CD because fewer disks are available to assume the workload of the failed disk. Therefore, the decrease in throughput in the event of a disk failure with ID is higher than with CD. With the MD scheme, the disk failure has little or no impact on the throughput because one or more disks are already fully utilized in normal mode.

Figure 6.26 presents the disk service time of all three schemes in both the normal and failure modes of operation. Figure 6.26a shows that, at a MPL of 1, the MD scheme provides the lowest disk service time (and thus highest throughput) among all three. The disk service time with MD increases as the MPL is pushed higher. This is because, as the MPL increases, both disk contention increases and the overhead of synchronizing two disk heads with write requests becomes higher. With the CD and ID schemes, the disk service time increases slightly when the MPL is increased from 1 to 2. When MPL > 2, the elevator algorithm used by CD and ID kicks in and the disk service time starts to decrease. As shown in Figure 6.26a, at a MPL of 24, the disk service time with MD is 55% higher than it is with CD. Figure 6.26 also shows that, with all three schemes, the difference in disk service time between the normal and failure modes is very small. This is because, as explained above, with all three schemes the workload increase due to the failure of a disk is small, and thus there is virtually no increase in the level of disk contention in the failure mode of operation. This experiment again demonstrates that for disk bound applications, the overhead of synchronous writes has a significant impact on system performance, whereas the overhead of remote update operations is negligible since the CPU overhead is overlapped with the disk service time.

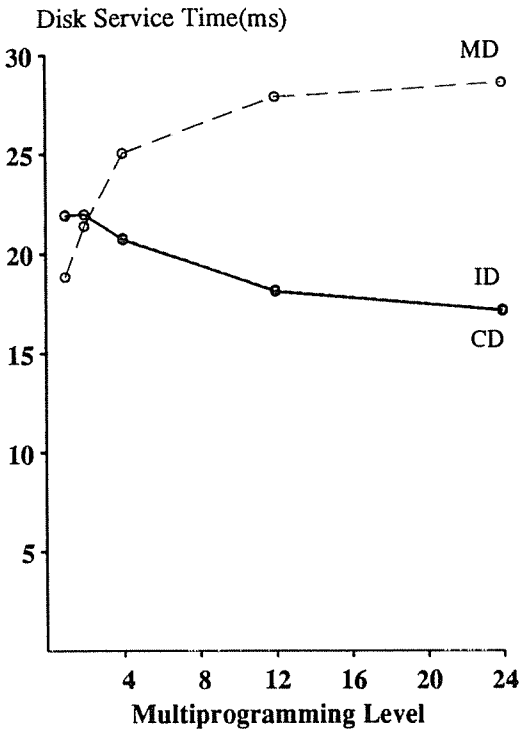

Figure 6.26a: 1% Update
Normal Mode

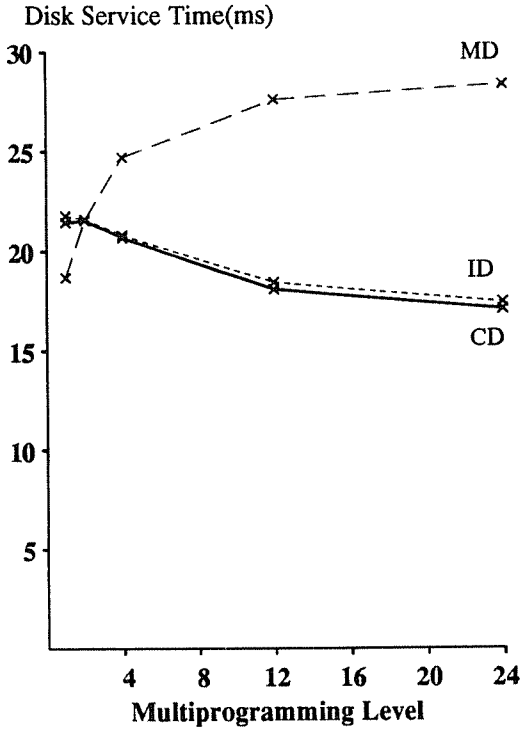

Figure 6.26b: 1% Update
Failure Mode

Experiment 6: 1% selection with X% update using a clustered index

As in Experiment 5, we assume here that either a relation is hash partitioned or a relation is range partitioned and the clustered index is constructed on a nonpartitioning attribute. Assume that the qualified attribute is not the range partitioning attribute, a 1% selection with X% update (X= 10, 30, or 50) query will have to be sent to all active nodes for processing in both the normal and failure modes of operation.

The first part of the query (1% selection) reads 1% of the tuples sequentially from each disk. The second part of the query (X% update) randomly updates from 10% to 50% of the tuples read in the first part and then writes the updated pages back to the local disk as well as the remote disk storing the backup copy. As in Experiment 2, a node with the CD and ID schemes reads and processes two more index pages and one more data page for each query than it does with MD in the normal mode of operation. While paying the overhead of reading more index and data pages, however, both CD and ID receive the benefit of higher intra query parallelism.

Figures 6.27 and 6.28 show, respectively, the throughput and disk service time of the 1% selection with X% update query type. In these two figures, suffixes _10, _30, and _50 are used to indicate 10%, 30%, and 50% update, respectively. In the normal mode of operation, Figure 6.27a shows that, overall, CD provides slightly higher

throughput than ID and much higher throughput than MD except at a MPL of 1. At a MPL of 1, both the CD and ID schemes provide higher throughput than the MD scheme when X is equal to 10, whereas MD provides higher throughput than CD and ID when X is equal to 30 or 50. The reason for the change in relative performance when the X changes is the following: At a MPL of 1, as described in Experiment 2, both disks at a node are actively reading matching tuples for a query with CD and ID, resulting in higher intra query parallelism, whereas only one disk at a time is actively reading disk pages with the MD scheme. On the other hand, as explained in Experiment 5, at a MPL of 1 disk writes originated from a remote node storing the primary copy will compete for disk service with disk reads/writes originated at a local node with CD and ID (resulting in a higher disk service time), whereas no disk contention occurs with MD. With CD and ID, the higher the update percentage is, the more severe the disk contention will be. When X = 10, only 10% of the tuples read are updated, and the overhead of higher disk service time (due to disk contention) with CD and ID is lower than the benefit of higher intra query parallelism. As a result, CD and ID provide higher throughput than MD there. When X = 30 or 50, however, the disk contention with CD and ID is much more severe, and the overhead of a longer disk service time outweighs the benefit of parallelism. As a result, MD provides higher throughput than CD or ID when X = 30 or 50. The effect of the update percentage on disk contention with CD and ID can be seen in Figure 6.28. At a MPL of 1, the disk service time with CD is 19.1 ms when X = 10 and 23.0 ms when X = 50. As illustrated in Figure 6.28, at a MPL of 1, the disk service time with CD and ID is about 2.6ms (15.8%) higher than with MD when X = 10 and 4.8ms (26.1%) higher when X = 50.

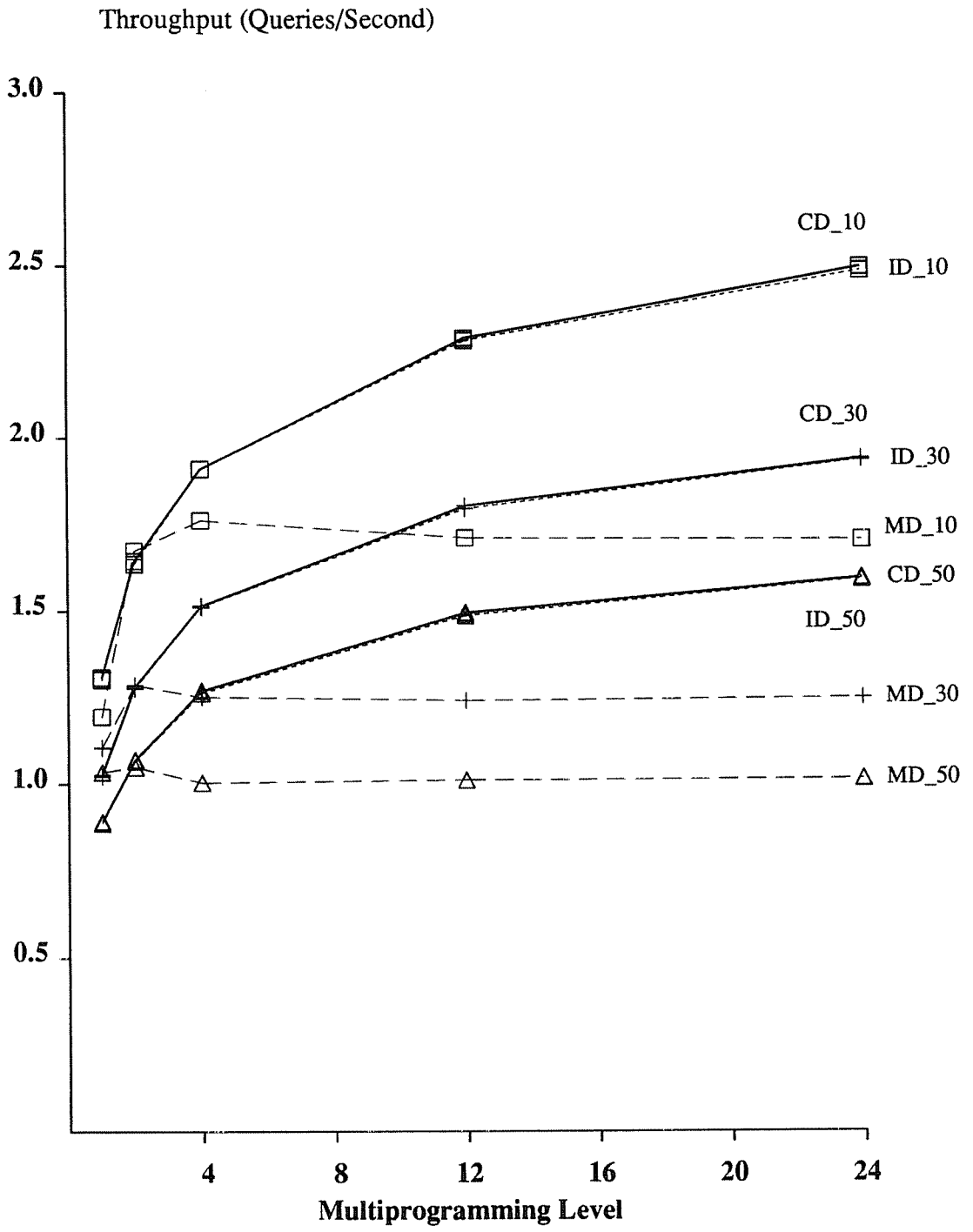

Figure 6.27a: 1% Selection with X% Update, No Failure

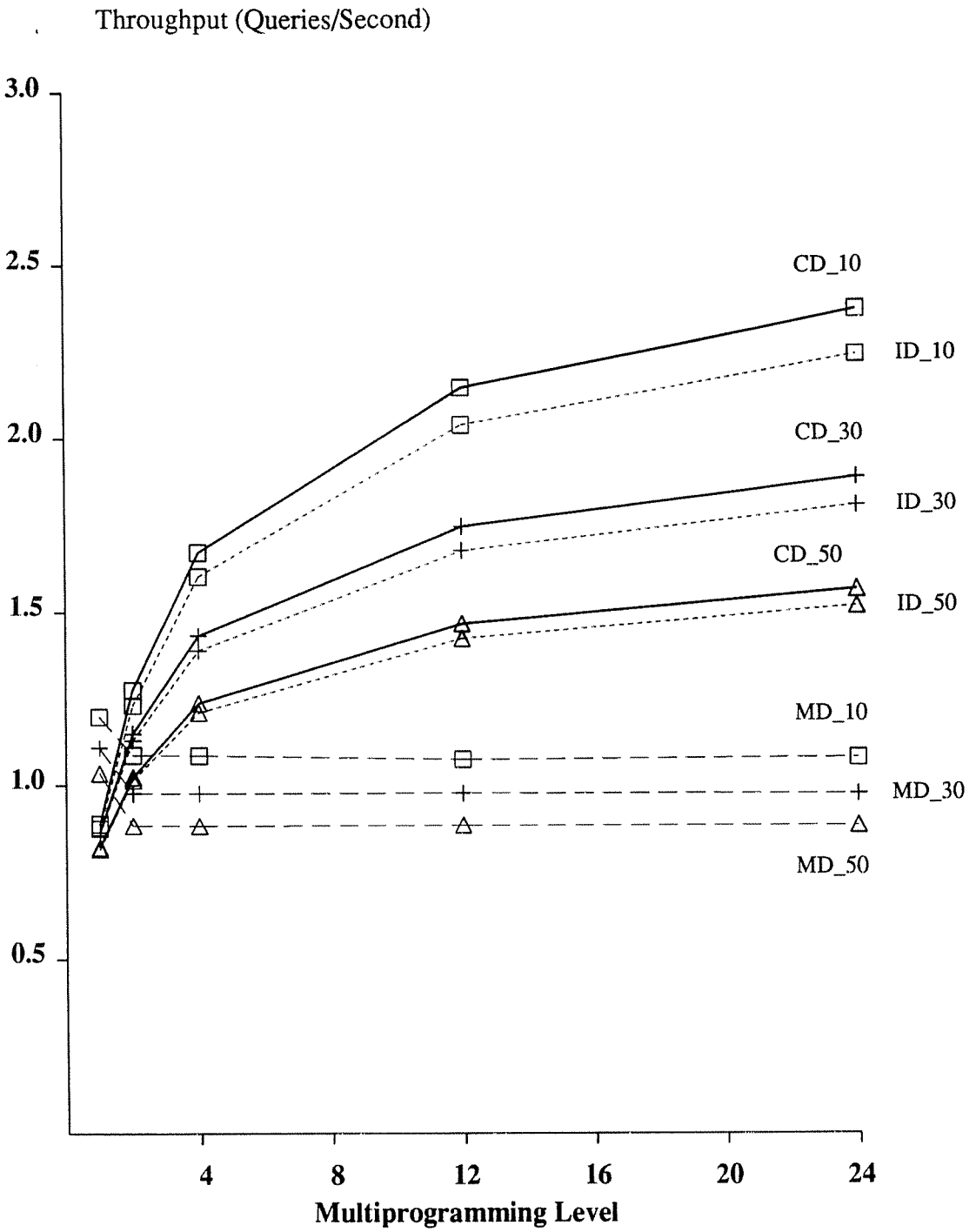

Figure 6.27b: 1% Selection with X% Update, Failure Mode

At a MPL of 2, queries from two distinct terminals (sources) will usually (but not always) be served by two different disks in a mirrored pair for retrieving tuples. Consequently, the update portion of the query will now cause disk contention with MD as well, resulting in higher disk service time. Similar to the case with CD and ID, the

higher the update percentage is, the more severe the disk contention will be. In addition, the synchronized disk write results in extra overhead with MD which is proportional to the update percentage. This is illustrated in Figure 6.28. At a MPL of 2, the disk service time with MD is 19.4 ms when X = 10, whereas it is 22.8 ms when X = 50. Figure 6.28 also shows that the disk service time with MD increases by 2.9ms and 4.5ms with X = 10 and 50, respectively, when the MPL is increased from 1 to 2. With CD and ID at a MPL of 2, not does only the update portion of a query result in contention at a disk, but the selection portion of two different queries may also compete for disk service resulting in an even higher disk service time when X = 10 and 30. However, the increase in disk service time is much smaller with CD and ID than with MD. When X = 50, the disk service time with CD and ID actually decreases slightly because there are enough disk requests in the queues to make the elevator algorithm effective.

Figure 6.28 shows that, at a MPL of 2, the disk service times with all three schemes are comparable when X = 50, whereas it is still lower with MD than with CD or ID when X = 10. One other factor affecting system throughput is intra query parallelism. Although not as significant as it is at a MPL of 1, the higher intra query parallelism with CD and ID still helps to generate higher throughput at a MPL of 2. As shown in Figure 6.27a, at a MPL of 2, MD provides slightly higher throughput than CD or ID when X = 10 and 30 but slightly lower throughput than CD or ID when X = 50. The difference in throughput is a combination of the effects of the difference in disk service time and the difference in intra query parallelism. For example, at a MPL of 2, when X = 10, the disk service time with CD is 13.6% higher than that with MD, but MD provides only 1.7% higher throughput than CD. This is because the benefit of higher parallelism with CD offsets some of the overhead of disk contention.

When X = 10, the throughput with MD continues to increase until a MPL of 4 is reached. When the MPL is greater than 4, though, one or more disks becomes fully utilized and forms a bottleneck with MD. The continuous increase in disk service times with MD results in a slight decrease in throughput from MPLs of 4 to 12. When MPL $\geq$ 12, the throughput with MD levels off. With X= 30 and 50, throughput decreases slightly from MPLs of 2 to 4 and levels off after MPL $\geq$ 4. The reason that the throughput with MD first decreases and then levels off is similar to that described in Experiment 5. With both the CD and ID schemes, throughput increases significantly from MPLs of 1 to 4 with all three update levels. When MPL > 4, the rate of increase drops significantly. This is because the disks are nearly fully utilized at a MPL of 4. The small increase in throughput when the MPL is greater than 4 is due mainly to the decrease in disk service time. At a MPL of 24, the CD scheme provides respectively 46%, 55%, and 57% higher throughput at X = 10, 30, and 50 than the MD scheme.

Figure 6.27b shows the throughput of all three schemes in the event of a disk failure. With the MD scheme, the disk failure has no impact on the throughput at an MPL of 1. The reason for this is the same as in Experiments 2 and 5. When the MPL is increased from 1 to 2, the throughput with MD decreases at all three levels of update percentage. This is because the remaining disk in the failed mirrored pair has a much higher disk service time due to disk contention between requests from different queries. Consequently, the disk in the failed mirrored pair becomes a bottleneck at a MPL of 2. The throughput with MD levels off when the MPL is greater than 2 because the disk in the failed mirrored pair is already fully utilized. As shown in Figure 6.27b, MD provides significantly lower throughput than CD for MPL > 4.

In the event of a disk failure, the throughput with CD and ID at a MPL of 1 drops by 32% when X = 10 whereas it drops only 8% when X = 50. The reason for the big performance degradation at X = 10 is that (as in Experiment 2) the workload of a system with CD and ID is not evenly redistributed on a per query basis in the event of a disk failure. For each query, one of the remaining disks serves almost twice as many requests as the other disks. In addition, the disk service time at the disk serving an extra subquery is higher than that at the remaining disks due to disk contention. As a result, each query takes much longer to complete in the failure mode than it does in the normal mode. There are three reasons that the drop in throughput at X = 50 is much smaller than it is at X = 10. First, there is less difference in disk service time among the remaining disks when X = 50 because the "remote" write requests were already competing with local requests for disk service. Second, while the number of disk requests at the disk serving an extra subquery is 72% higher at X = 50 than that at each of the other remaining disks, it is 90% higher at X = 10. Finally, the node serving an extra subquery may not have to initiate the "remote" writes if its corresponding "remote" node happens to have the failed disk connected to it. In that case, the node serving an extra subquery will have less CPU overhead and there will be no need for the query to wait for "remote" writes to complete. With both the CD and ID schemes, the drop in throughput becomes less drastic for MPL > 4. This is because the workload is more evenly distributed among the remaining disks at high MPLs. At a MPL of 24, the throughput drops by less than 5% with CD and less than 10% with ID when a disk failure occurs.

Figure 6.27b also illustrates that CD provides higher throughput than ID throughout the entire range of MPLs in the failure mode of operation. This is because the CD scheme provides better load balancing than the ID scheme does as explained earlier, while both schemes incur about the same level of overhead in the event of a disk failure.

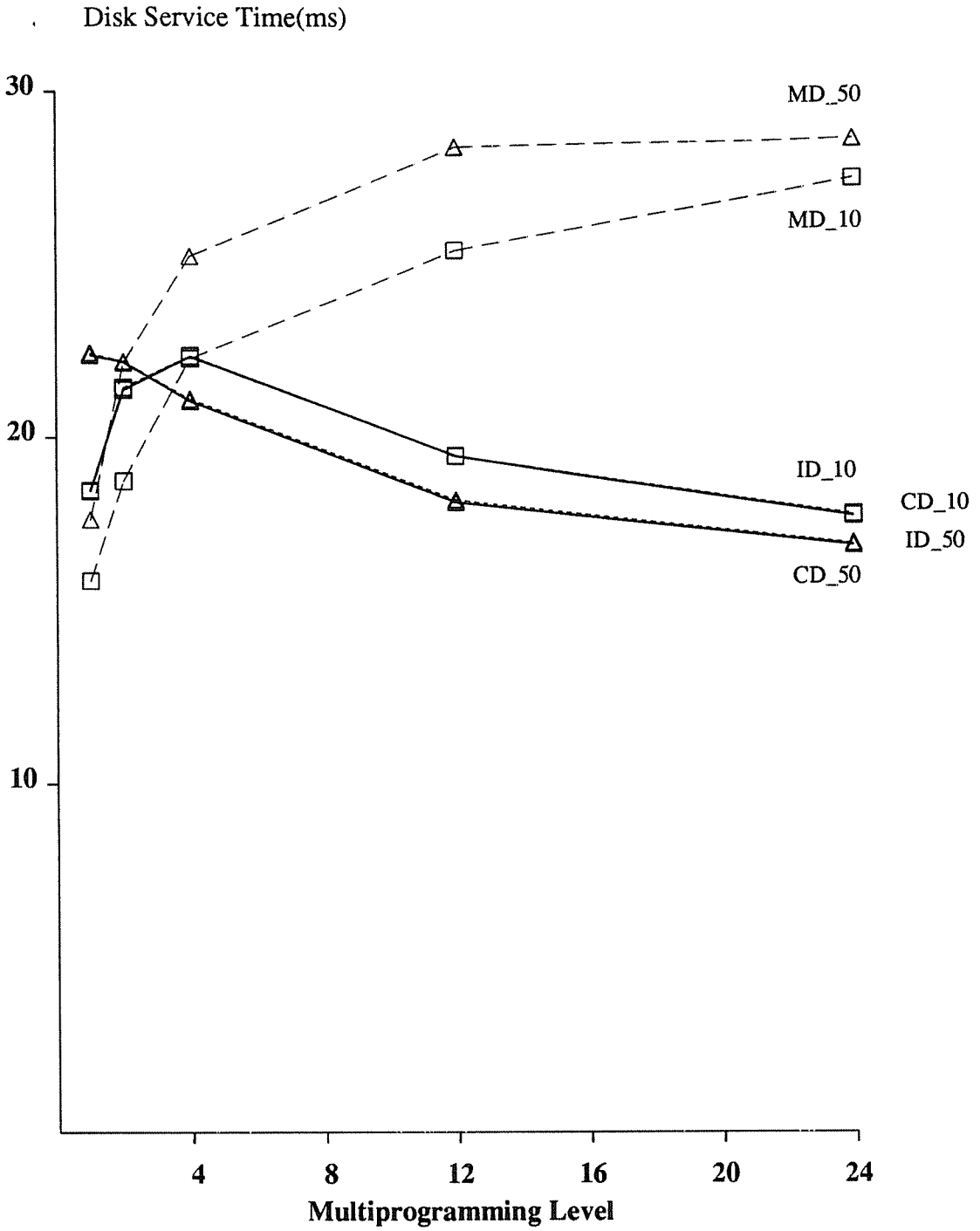

Figure 6.28: 1% Selection with X% Update, No Failure

6.4.3. Varying CPU Speed and/or Page Size (Experiments 7 - 9)

All of the results that have been presented so far were based on a 3 MIPS CPU and an 8K disk page size. In this section, we present some performance results obtained using different combinations of CPU speeds and page

sizes. The purpose of this study is to investigate how changing the CPU and/or page size parameters affects system performance with the different replication schemes. In this section, we repeat the same test using a higher CPU speed, using 14 MIPS CPU instead of 3 MIPS, if an application (i.e., a query type) is CPU bound. If an application is I/O bound, however, a higher CPU speed will not affect the relative performance of the three different replication schemes. In such cases, the test is instead rerun with the page size reduced from 8K to 4K bytes while the CPU speed is kept the same.

Experiment 7: Single tuple selection on the partitioning attribute (4K page)

In this experiment, the page size parameter is set to 4K. By reducing the page size from 8K (as in Experiment 1) to 4K, a disk is able to serve requests with a lower transfer time, but the number of CPU cycles needed to process a page is also reduced. Because the decrease in disk service time per page is not significantly higher than the decrease in CPU processing time for a page, this query type remains I/O bound with a 3 MIPS CPU and a 4K page size. Figure 6.29 shows the throughput of a single tuple selection query on the partitioning attribute using an index (either clustered or nonclustered). As one can see from this figure, the CD and ID schemes still provide higher throughput than the MD scheme in the normal mode of operation, and CD continues to provide higher throughput than the other two schemes in the event of a failure.

As shown in Figure 6.29, the relative performance of the three replication schemes in both the normal and failure modes of operation are the same as those in Figure 6.3. However, the throughputs of all the three schemes in this experiment are lower than their counterparts in Experiment 1 even though each data/index page (4K in size) in this experiment requires less disk and CPU processing than each 8K page did in Experiment 1. The reason for this "strange' behavior is the following: The depth of a B+ index tree is 3 levels in this experiment, whereas it is only 2 in Experiment 1. A single tuple selection query using an index therefore accesses more index pages here than it does in Experiment 1. Because the overhead of reading and processing one extra index page for each query is higher than the savings in disk transfer time and CPU processing time for each page, the throughput of all three schemes decreases when the page size is reduced from 8K to 4K bytes.

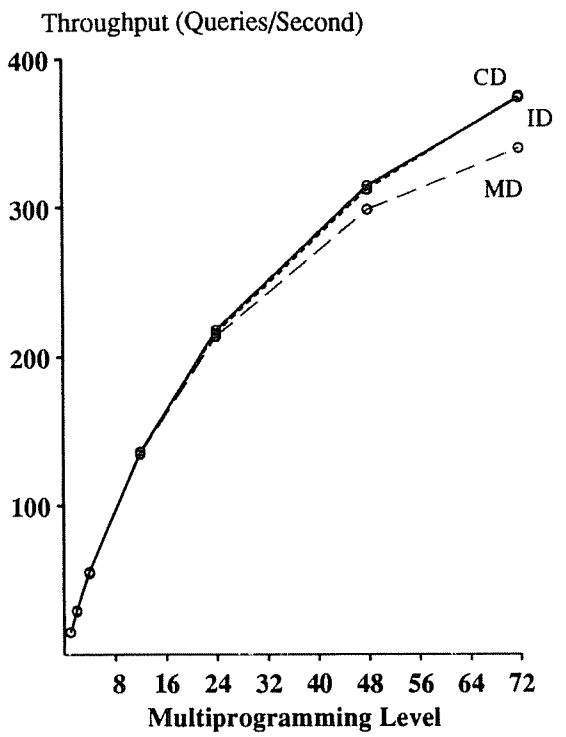

Figure 6.29a: Single Tuple Selection
Normal Mode

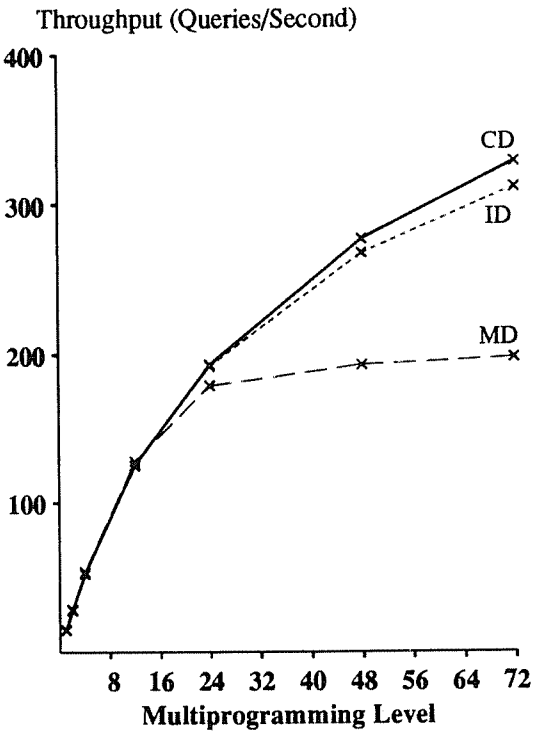

Figure 6.29b: Single Tuple Selection
Failure Mode

We also ran the same query type (single tuple selection) with a 14 MIPS CPU and a 4K page configuration. The results of this experiment were very similar to those for the 3 MIPS CPU and 4K page configuration.

Experiment 8: Single tuple update query (4K page)

In this experiment, the page size is again lowered to 4K. Similar to Experiment 7, a single tuple update query remains I/O bound when the disk page size is reduced from 8K (as in Experiment 4) to 4K. Figure 6.30 shows the performance results of a single tuple update query using an index. At low multiprogramming levels (MPL $\leq$ 12), the MD scheme provides a slightly higher throughput than the other two schemes in both modes of operation but the difference in throughput is much smaller than that in Experiment 4 (Figure 6.18). On the other hand, both CD and ID provide significantly higher throughput than MD when the MPL is greater than 24. Figure 6.30b also shows that CD provides slightly better performance (5% higher throughput at a MPL of 72) than ID when a disk/node failure occurs. Overall, the throughput curves in Figure 6.30 are similar to those in Figure 6.18.

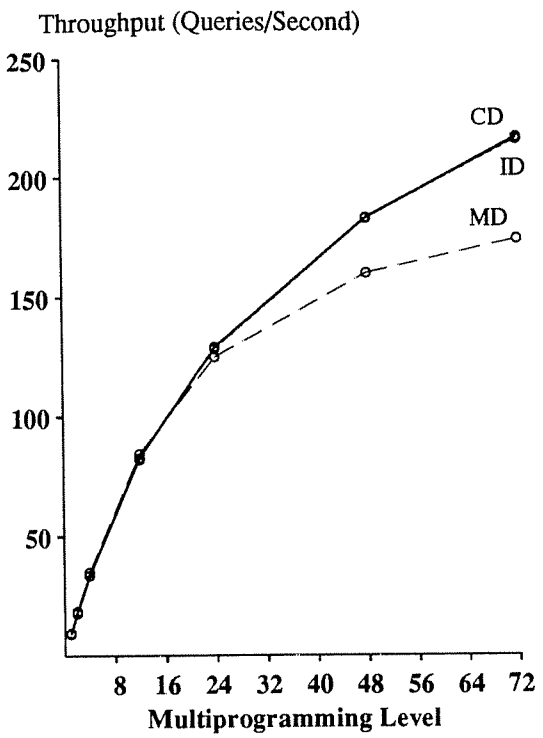

Figure 6.30a: Single Tuple Update
Normal Mode

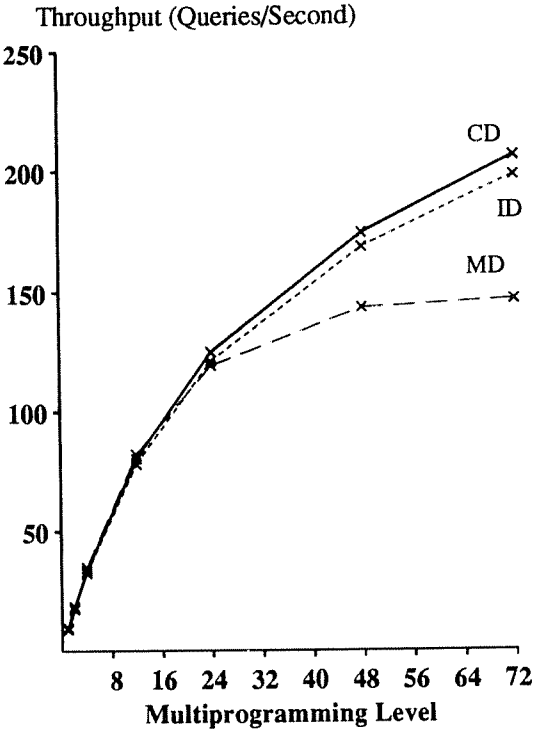

Figure 6.30b: Single Tuple Update
Failure Mode

We also ran the single tuple update query with a 14 MIPS CPU and 4K page configuration. The results of this experiment are presented in Figure 6.31. The major difference between the results in Figures 6.31 and 6.30 is that the MD scheme no longer provides higher throughput than the CD scheme (the difference is unnoticeable) at low multiprogramming levels. This is because the CPU overhead of packaging, sending, and receiving an updated page with CD is so small with a 14 MIPS CPU that it has little or no impact on system performance.

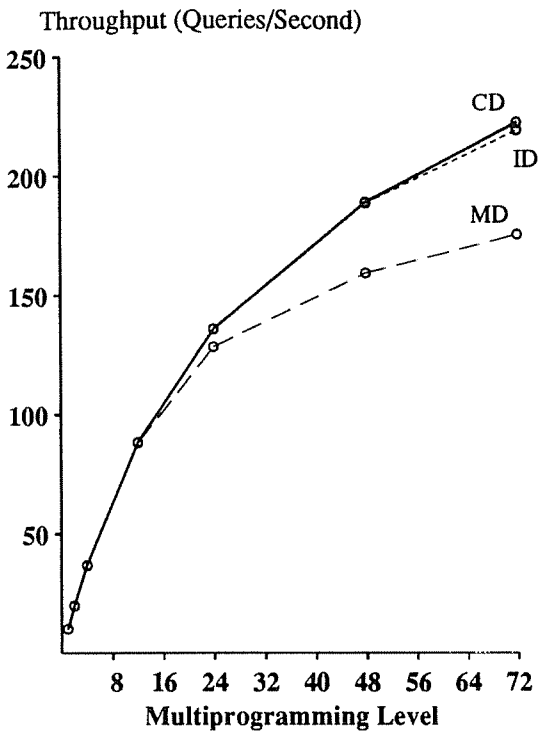

Figure 6.31a: Single Tuple Update
Normal Mode

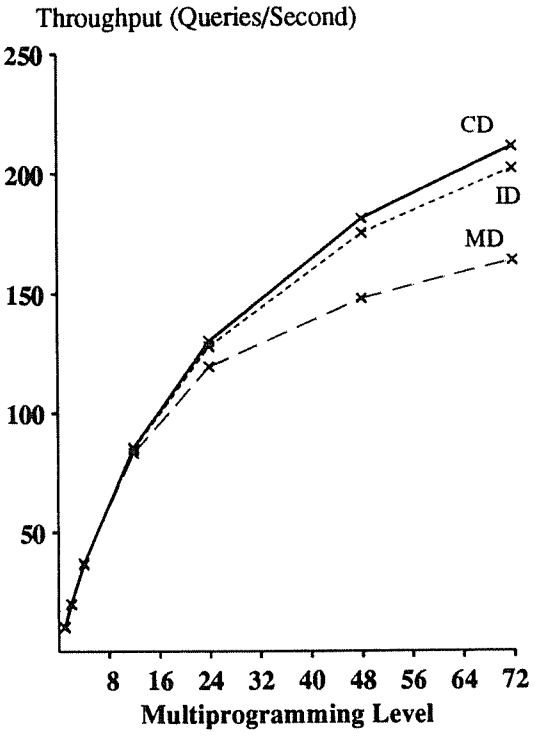

Figure 6.31b: Single Tuple Update
Failure Mode

Experiments 7 and 8 demonstrate that, for a single tuple selection query, varying the CPU speed and/or the page size does not change the relative performance of the three schemes in any noticeable way. For a single tuple update query, on the other hand, a faster CPU and/or a smaller page size favor both the CD and ID schemes because the overhead of updating remote backup copy decreases as either the CPU speed is increased or the page size is decreased.

Experiment 9: 1% selection query using a clustered index (14 MIPS and 8K page)

With a 3 MIPS CPU and an 8K bytes data page, a 1% selection query using a clustered index was shown to be CPU bound in Experiment 2. To study how a faster CPU affects the performance of each of the three replication schemes, the model was configured to have a 14 MIPS CPU and an 8K byte data page size. Figure 6.32 shows the throughput of this query with this new configuration.

Comparing Figures 6.32 and 6.11 (of Experiment 2), one can see that increasing the CPU speed from 3 to 14 MIPS significantly changes the relative performance of the three replication schemes. Figure 6.32a shows that, with a faster CPU, the MD scheme no longer performs better than CD or ID at high MPLs in the normal mode of opera-

tion. For example, at a MPL of 48, CD and ID provide 32% higher throughput than MD in this experiment (compared with Figure 6.11a, where MD provided 5% higher throughput than CD and ID at a MPL of 48). The reason for this change is that the query is I/O bound and not CPU bound in this experiment. Consequently, the CPU overhead of activating more selection processes and processing more index pages with CD and ID has little or no impact on the system throughput while the benefit of higher intra query parallelism with CD and ID becomes significant. In this experiment, the higher throughput exhibited by CD and ID in the normal mode is the result of higher intra query parallelism and the difference in disk scheduling algorithms, as explained in Experiment 2.

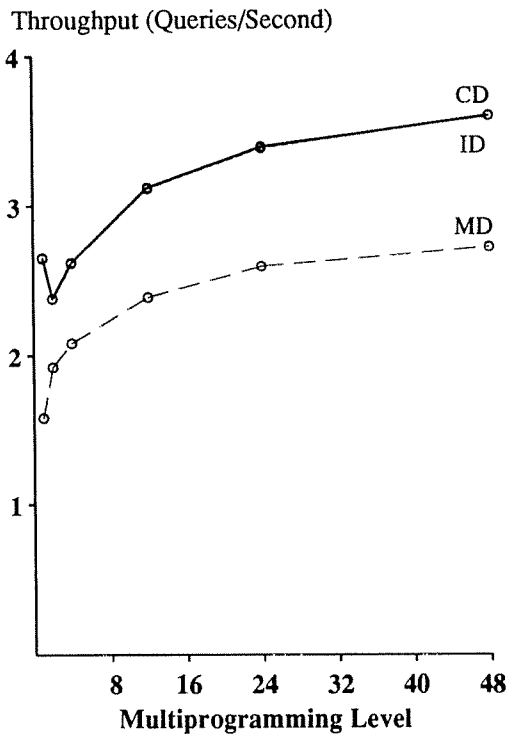

Figure 6.32a: 1% Selection
Normal Mode

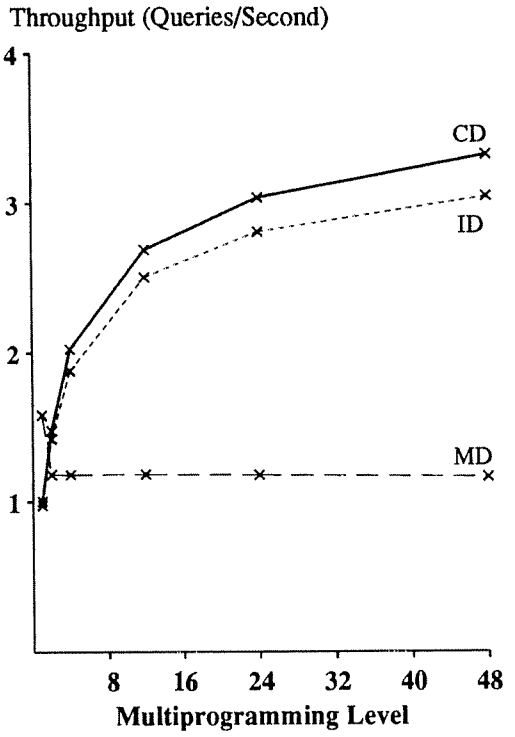

Figure 6.32b: 1% Selection
Failure Mode

There is one other major difference between Figures 6.32a and 6.11a; when the MPL is increased from 1 to 2, Figure 6.32a shows that the throughput decreases rather than increases with CD and ID. This is because, for this experiment, the savings from overlapping CPU and disk operations at a MPL of 2 is lower than the increase in disk service time; with a 14 MIPS CPU, the CPU processing time is only a small fraction of the total query execution time and thus savings is also small. With the MD scheme, the throughput does not decrease when the MPL is increased from 1 to 2. This is because, at a MPL of 2, two different queries are likely to be served by two distinct

disks within the same mirrored pair and thus the increase in disk service time (due to disk contention) with MD is small.

In the event of a disk failure, Figure 6.32b shows that both CD and ID again provide significantly higher throughput than MD (similar to Figure 6.11b). In addition, Figure 6.32b shows that, at a MPL of 48, the throughput with CD is 9.2% higher than that with ID. However, the difference in throughput between CD and ID in this experiment (9.2% at a MPL of 48) is larger than in Figure 6.11b (4%). The reason for this difference is that the average disk load in the failed ID cluster is about 14% higher than in the normal clusters, whereas the CPU load of a node in the failed ID cluster is only about 7% higher than the remaining processors (since every processor has two disks connected to it and at most one of its disks is operating in failure mode). Because a 1% selection query is I/O bound in this experiment, whereas it was CPU bound in Experiment 2, the performance degradation with ID is higher in this experiment than in Experiment 2. In contrast, with the CD scheme the workload of a disk or a processor increases by only about 3% when a disk failure occurs. As a result, the performance degradation with CD is smaller in both experiments.

One other difference between Figures 6.32b and 6.11b is that the throughput with MD decreases from a MPL of 1 to 2 in Figure 6.32b because, at a MPL of 2, two queries compete for disk service on the remaining disk in the failed mirrored pair. This results in a much higher disk service time on that disk. Consequently, the remaining disk in the failed mirrored pair becomes a bottleneck and the throughput of the system drops.

Other Experiments:

We also reran the 0.1% selection query (Experiment 3) and the 1% update query (Experiment 5) with different combinations of CPU speed (3 MIPS or 14 MIPS) and disk page size (4K or 8K). Both query types remained I/O bound. The results of those experiments are similar to those presented in Experiments 3 and 5, so we will not discuss them further here.

6.5. Summary

Our results indicate that chained declustering provides much better performance than mirrored disks in the event of a disk failure. In addition, when a failure has not occurred, chained declustering can potentially provide better performance than MD for I/O bound applications, and performance comparable to mirrored disks for CPU bound applications. As stated earlier, advances in CPU technology have tended to be much faster than those of disk

technology [Joy85, Fran87], so we believe that most database applications are/will be likely to be I/O bound.

The interleaved declustering scheme provides performance comparable to the chained declustering scheme for read only applications in the normal mode of operation. For applications that require both disk reads and writes, though, interleaved declustering provides slightly lower throughput than CD when no failures have occurred. On the other hand, when a disk/node failure occurs, interleaved declustering provides noticeably lower throughput than chained declustering due to its limited ability to balance the workload and to the overhead of activating extra processes and processing extra index pages for certain query types.

CHAPTER 7

TRANSACTION AND FAILURE MANAGEMENT

7.1. Introduction

Transaction and failure management, which manages initialization, execution, termination and recovery of all database transactions, is one of the major components of a database management system. The primary goal of transaction management is to ensure synchronization atomicity and failure atomicity which together provide a consistent view of the database to all application programs. By synchronization atomicity we mean that the concurrent execution of multiple transactions has the same effect as if they were executed serially, and hence they do not interfere with each other's operation. By failure atomicity we mean that the execution of a transaction is "all or nothing": either all or none of its effects are in the database. How to provide synchronization atomicity is commonly referred to as the concurrency control problem, and how to enforce failure atomicity is commonly referred to as the recovery problem.

In the preceding chapters, we have presented the design and evaluation of the chained declustering data replication scheme. In this chapter, we will explore issues related to transaction and failure management when the chained declustering strategy is employed. In particular, we will investigate how the mechanisms that Gamma currently uses for transaction and failure management[1] (which are based on two-phase locking [Gray78] and the write ahead log (WAL) protocol [Gray78]) can be augmented or modified to work with the chained declustering strategy.

7.1. Failure Management with Chained Declustering

In broad terms, a recovery technique must handle three types of failures: transaction failures, including both user aborts and system aborts (e.g., deadlock victims); system failures resulting from processor, memory and other electronic component failures; and media (disk) failures. In a non-replicated database, the log-and-checkpoint

[1] The implementation of the log based recovery algorithm has not been completed yet.

protocol [Gray78, Lind81] has been widely adopted for recovering from all three types of failures.

For multiprocessor shared-nothing database machines, implementation of the log-and-checkpoint based recovery mechanism can take one of two basic approaches: the distributed log approach and the log server approach. With the distributed log approach, log records generated at a node are stored on a disk (or disks) attached to the same node. With the log server approach, a node is designated as the log server and log records generated at all query processing nodes are sent to the server for storage. For a system with a large number of nodes, more than one log server may be required to avoid having the server becomes a bottleneck. Currently, both the Gamma and Tandem NonStop SQL database machines adopt the log server approach [DeWi90, Tand88].

With replicated data, a single failure of any hardware component (processor or disk) will not compromise the availability of data. Occasionally, however, two or more components of the full system may fail simultaneously due to power failure, human error, or other causes. To ensure database consistency in such an event, a recovery scheme based on logs (or some other mechanism) is still needed. With chained declustering, the two log based recovery approaches can continue to be used with minor modifications. If the distributed log approach is used, the primary and backup copies of a relation can be treated as two independent relations, and each node can generate and maintain its own update log records when data items stored on the node are modified. If the log server approach is used, update log records will be generated only at nodes holding the primary fragments (or at the nodes where subqueries are processed) and the log records will be sent to the log server for storage. In this case, both the primary and backup copy nodes will share the same update log records and use them to recover from failure if necessary.

Currently, the recovery algorithms planned for Gamma (for a non-replicated database) operate as follows. When an operator process updates a record, it also generates a log record which records the change in the database state. Associated with every log record is a log sequence number (LSN) which is composed of a node number and a local sequence number. The node number is statically determined at system configuration time, whereas the local sequence number, termed **current LSN**, is a monotonically increasing value.

Log records are sent by the query processors to the Log Manager/log server (running on a dedicated processor with its own disks) which merges the log records received from all query processors to form a single log stream. When a page of log records is filled, it is written to disk. The Log Manager maintains a table, called the **Flushed Log** table, which contains, for each node, the LSN of the last log record from that node that has been flushed to disk. These values are returned to the nodes either upon request or when they can be piggybacked on another message.

Query processing nodes save this information in a local variable, termed the **Flushed LSN**.

The buffer managers at the query processing nodes observe the WAL protocol [Gray78]. When a dirty page needs to be forced to disk, the buffer manager first compares the page's LSN with the local value of Flushed LSN. If the page LSN of a page is smaller than or equal to the Flushed LSN, that page can be safely written to disk. Otherwise, either a different dirty page must be selected, or a message must be sent to the Log Manager to flush the corresponding log record(s) of the dirty page. Only after the Log Manager acknowledges that the log record has been written to the log disk will the dirty data page be written back to disk.

The query scheduler is responsible for sending commit or abort records to the Log Manager. If a transaction completes successfully, a commit record for the transaction is generated by its scheduler and sent to the Log Manager, which employs a group commit protocol [DeWi84]. On the other hand, if a transaction is aborted by either the system or the user, its scheduler will send an abort message to all query processors that participated in its execution. The recovery process at each of the participating nodes responds by requesting the relevant log records generated by the node from the Log Manager (recall that the LSN of each log record contains the originating node number). As the log records are received, the recovery process undoes the log records in reverse chronological order using the ARIES undo algorithm [Moha89]. The ARIES recovery methods are also used as the basis for checkpointing and restart recovery.

When the chained declustering scheme is implemented on Gamma, this same recovery algorithm can continue to be used for transaction commit and abort. When an operator process completes the processing of a transaction, it sends an "operator done" message to the corresponding scheduler. Upon receiving "operator done" messages from all participating operator processes, the scheduler commits the transaction.

When an operator process fails to complete the processing of a transaction, the transaction will be aborted. To abort a transaction, each of the participating nodes will request the log records generated by the node from the Log Manager (log server). As the log records are received, each node undoes the updates by the aborted transaction using the same algorithm currently used in Gamma. The undone pages can then be sent to the corresponding backup copy node to undo the effects of the aborted transaction there. As an optimization, if updated pages received from the primary copy node are not written to disk at the backup copy node until transaction commit time (i.e., if there are enough buffer pages), a backup copy node can discard those updated pages generated by the aborted transaction (which are still in the buffer) upon receiving an abort message from the Scheduler. In this case,

the undone pages will not have to be sent from the primary copy node to its backup copy node when a transaction is aborted. In the event of a node (processor or disk) failure that renders a disk (or disks) inaccessible, the following two approaches can be used to restore database consistency after the failed component is repaired or replaced: The first approach is to spool and then apply the redo log records to redo all missed updates at the recovering node [Hamm80], and the second approach is to use a *copier* transaction to restore the failed disk [Atta84, Bern84, Bhar86]. In the event of multiple failures that render both copies of one or more relations unavailable, an analysis phase will be needed to determine the winner and loser transactions at recovery time (when the failed components are repaired or replaced) and the log will be used to redo the updates of the winner transactions and to undo the updates of the loser transactions. In such a case, the corresponding log records can be used to restore the primary copies, and then the corresponding backup copies can be brought up to date by a copier transaction. To further speed up log based recovery, an update log record could contain the node numbers of both the primary copy and backup copy nodes. At recovery time, the log record could then be sent simultaneously to both nodes for parallelism during recovery.

7.2. Transaction Management with Chained Declustering

Concurrency control mechanisms in both centralized and distributed DBMSs are generally based on one of three basic approaches: locking [Mena78, Gray78, Lind81], timestamp ordering [Thom79, Bern80, Reed83], and optimistic concurrency control [Kung81, Schl81, Ceri82]. Currently, concurrency control for non-replicated databases in Gamma is based on two-phase locking [Gray78]. Each node in Gamma has its own local lock manager and deadlock detector. The lock manager maintains a lock table, which is implemented as a chained bucket hash table, and a transaction wait-for graph. An entry in the lock table is selected by hashing on either a page-id or a file-id. The transaction wait-for graph is also a chained bucket hash table, and is organized by hashing on the transaction id. Besides being used for deadlock detection, the wait-for graph maintains the list of resources acquired by a transaction during its lifetime. This allows for the efficient release of locks at the commit point of the transaction.

When a transaction attempts to set a lock on either a page or a file, the lock manager first checks the lock table to see whether another transaction already has a lock on the object. If not, the lock manager creates an entry in the lock table corresponding to the resource and records the fact that this transaction has now acquired this resource. If a lock on the requested resource does already exist in the lock table, then the lock manager will check the compatibility of the current lock(s) on the resource with the pending lock request. If the pending lock request is compatible,

the lock manager records the additional lock-holder in the entry in the lock table and records the lock in the requesting transaction's cell (node) in the wait-for graph. If the pending lock request is incompatible, the lock manager first determines whether a deadlock will occur if the transaction requesting the lock is allowed to wait. If a cycle in the wait-for graph is found, the requesting transaction is aborted. Otherwise, the transaction is blocked until either the owner of the lock terminates or the mode of the granted group changes.

Gamma uses a centralized deadlock detection algorithm to detect multi-node deadlocks. Periodically, the centralized deadlock detector sends a message to each node in the configuration, requesting the local transaction wait-for graph of that node. After collecting the wait-for graph from each node, the centralized deadlock detector creates a global transaction wait-for graph. Whenever a cycle is detected in the global wait-for graph, the centralized deadlock manager chooses to abort the transaction holding the fewest number of locks.

With chained declustering, if the primary copy access (locking) scheme is used, the concurrency control mechanism described above can continue to be used without any modifications in the normal mode of operation. With no failed components, a transaction will be directed to the node holding the primary fragment. It will request access permission (locks) locally at the nodes storing the relevant primary copies. After obtaining access permission, the transaction manipulates the requested data locally. Of course, any write operations must also update the backup copies at some point prior to transaction commit. However, no new locks must be acquired. An exclusive (write) lock on the primary copy is sufficient to guarantee data consistency and serializability of transactions since the backup copy of a page or file is not used for processing queries in the normal mode of operation.

When chained declustering is used in conjunction with the load balancing algorithms described in Chapter 3, both the primary and backup copies of a relation are used for processing queries in the failure mode of operation. When operating in the failure mode, transactions could continue to acquire locks from the nodes storing the primary copies[2] (the primary copy locking scheme) before they manipulate data. With this locking scheme, a transaction running in the failure mode may need to request lock(s) from a remote node (where the primary copy is stored) before accessing data stored locally. Because a remote lock request is much more expensive than a local lock request, the primary copy locking scheme may not be a good strategy in the failure mode of operation.

---

[2] For the purpose of setting locks, the corresponding backup fragment of a failed primary fragment is "promoted" to become the primary fragment in the event of a node failure.

There are, however, several more efficient alternatives for locking in the failure mode of operation. One alternative is to set read locks at the nodes containing the accessed pages and write locks at both the primary and backup copy nodes (i.e., lock both copies for a write operation in the failure mode). With this strategy, a subquery will only have to request locks locally before reading or writing any data pages. The write lock at remote copy can be deferred until transaction commit time [Care89]. The overhead associated with this strategy is that a subquery needs to lock both the primary and backup copies in the case of a write operation. However, since the cost of setting a lock locally in Gamma ranges from 100 to 250 instructions [DeWi90], the overhead of setting one extra write lock in the failure mode of operation should not noticeably affect the system performance.

The other alternative is to apply logical update operations at both the primary and backup copies (as is done by Teradata) rather than sending updated pages across the communication network. With this strategy, each node sets locks locally for both local read and local write accesses to a page. However, as described in Chapter 5, this strategy incurs almost twice as many disk accesses as the strategy that sends updated pages through the communication network. Unless the network bandwidth is low and applications are highly CPU bound, this strategy will not be an attractive alternative.

CHAPTER 8

CONCLUSION AND FUTURE RESEARCH DIRECTIONS

8.1. Summary

Motivated by the need to provide both high availability and high performance in a database system, we initiated a study of the problem of data replication in database machines. To benefit from the results of previous related studies, we conducted a review of the literature in the area of high availability strategies. The existing high availability strategies are based on two general approaches: the identical copy approach and the disk array approach. Two studies by Gray [Gray90] and Chen [Chen90] have shown that mirrored disks (an identical copy based strategy) provides better performance than RAID (a disk array based approach) for OLTP applications and for applications that transfer less than one track of data per request. Because an I/O request in most database applications is likely to transfer less than one track of data, the identical copy approach is potentially more suitable for database applications than the disk array approach.

Two identical copy based high availability strategies, namely mirrored disks and interleaved declustering, have been used commercially. Although both strategies significantly improve the probability that data remains available in the event of a failure, each has one or more significant limitations. The most significant limitation is that neither of these schemes can provide both high availability and high performance when failures occur. The search for a better solution led us to design a new high availability strategy termed chained declustering.

In Chapter 3, we presented the chained declustering scheme and the strategy used by chained declustering to balance the load in the event of a failure. The keys to the solution of balancing the workload are the notion of a responsible range for index attributes, the availability of an extent map for relations stored as a heap, and the use of query modification techniques [Ston75]. Three load balancing algorithms were presented in Chapter 3 that can be used to handle all combinations of partitioning methods, storage organizations and access plans found in typical parallel database systems [Tera85, DeWi86, Engl89].

Using a simple analytical model, we studied in Chapter 4 the degree of data availability provided by different high availability schemes and the ability of each scheme to balance the workload in the event of a node failure. Our results showed that the mirrored disk scheme provides the highest data availability while the chained declustering scheme is second (providing half the availability of mirrored disks). One significant drawback of the mirrored disk scheme, however, is that it cannot balance the workload in the event of a node failure. In the worst case, the load increase on a node can be as high as 100% when its neighboring node fails. In contrast, the chained declustering scheme can evenly redistribute the workload of a failed node among all remaining nodes in addition to providing high data availability. The interleaved declustering scheme provides a tradeoff between availability and performance in the event of a failure; as its cluster size is increased, the probability of two failures rendering data unavailable increases while the imbalance in workloads among the processors in the event of a failure decreases.

To better understand the relative performance of different data replication schemes in various multi-user situations, we implemented a simulation model for each of the identical copy schemes studied. The simulation model is based on the Hypercube version of the Gamma database machine. Chapter 5 described the components of our simulation model. Each component of the simulation model is implemented by a DeNet [Livn89] discrete event module.

In Chapter 6, we studied the performance of the chained declustering, interleaved declustering, and mirrored disk schemes using the simulation model described in Chapter 5. We first validated the model with results obtained from the Gamma database machine. The results produced by the simulation model were shown to accurately match Gamma's actual performance. We then ran performance experiments using both read only selection queries and update queries requiring both reads and writes.

For selection queries, chained declustering and interleaved declustering were shown to perform comparably in the normal mode of operation. Both performed better than mirrored disks if an application is I/O bound (due to disk scheduling), but slightly worse than mirrored disks if the application is CPU bound. In the event of a failure, chained declustering was indeed able to balance the workload among the remaining disks, while interleaved declustering redistributed the workload within the failed cluster; mirrored disks cannot do any load redistribution, so the mirror image of the failed disk had to process all requests originally served by the failed disk. As a result, chained declustering provided slightly better performance than interleaved declustering and much better performance than mirrored disks in a failure mode of operation.

To update the backup copy of a data item, chained declustering and interleaved declustering incur the CPU overhead of packaging and sending the updated data to the remote node where the backup copy is stored. In addition, the remote node consumes extra CPU cycles to receive the network packet (containing the updated page) and to initiate an extra disk write operation to write the updated page to disk. With mirrored disks, both copies of a data item are stored on disks attached to the same processor. Consequently, no extra CPU cycles are needed for updating the backup copy. However, with mirrored disks each write operation ends up synchronizing the read/write heads of both disks in the mirrored pair. Therefore, the disk service time for a write becomes the maximum service time of two writes. In addition, both disk arms in a mirrored pair will be at the same cylinder after each write operation, effectively reducing the number of disk arms available for serving the next read request to one. Consequently, with mirrored disks, the average disk service time per request is longer for update queries than for select queries.

For update queries, because chained declustering and interleaved declustering incur CPU overhead while mirrored disks incurs overhead in disk seek distance, the relative performance of the three schemes depends on the relative performance of the processor and the disk drive of a system. In the normal mode of operation, if an update query is I/O bound, chained declustering and interleaved declustering perform better than mirrored disks. On the other hand, if an update query is CPU bound, the mirrored disks scheme will perform somewhat better than the other two schemes. However, because advances in CPU technology occur much faster than those of disk drive technology, we believe that most future database applications will be disk bound. Consequently, chained declustering should provide better performance than mirrored disks even with update intensive applications.

When failures occur in the mirrored disk scheme, a bottleneck forms at the failed mirrored pair; throughput is then limited by the rate at which the failed pair can service requests. On the other hand, with chained declustering the workload of a failed disk is again evenly redistributed among the remaining disks. Consequently, chained declustering provides much higher throughput than mirrored disks in the event of a failure. The relative performance of chained declustering and interleaved declustering (ID) in the event of a disk failure depends on the query type and the size of an ID cluster. With an ID cluster size of 8, our experiments showed that chained declustering can provide as much as 14% more throughput than interleaved declustering for a single tuple update query and as little as a 3% improvement for a 1% update query. Notice, however, that with an ID cluster size of 8, besides providing lower throughput, the interleaved declustering scheme is 3.5 times more likely to have unavailable data than is the chained declustering scheme.

In Chapter 7, we investigated issues related to the integration of the chained declustering data replication scheme and the transaction and failure management component of a DBMS. With chained declustering, a single failure of any component will not compromise the availability of data. Occasionally, however, two or more components of the full system may fail at the same time due to power failure, human error, or other causes. To ensure database consistency in such events, a recovery scheme based on logs (or some other mechanism) is still needed. When the chained declustering data replication scheme is employed in Gamma, the log based recovery protocol that Gamma currently uses for a non-replicated database can continue to be used. In addition, the concurrency control mechanism currently implemented in Gamma, which is based on two-phase locking, can also continue to be used in the normal mode of operation when a primary copy locking scheme is used with chained declustering. In the event of a failure, the current locking protocol used by Gamma may not be sufficient and some modifications may be needed in order to ensure synchronization atomicity.

8.2. Conclusion

In this dissertation we have introduced a new declustering technique, termed chained declustering, for distributing replicated data in a shared-nothing multiprocessor database machine. Chained declustering was shown to provide a very high degree of availability, second only to Tandem's mirrored disk approach. Although current disk technology makes the simultaneous failure of two disks very unlikely, with very large database machine configurations such failures will indeed occur occasionally.

The major contribution of chained declustering is not, however, its robustness with respect to two simultaneous node (processor or disk) failures. Rather, it is the ability of chained declustering to equally balance the workload among the remaining processors when a failure occurs. While Teradata's replication technique can also uniformly distribute the workload in the event of a failure, if any two nodes in a Teradata cluster fail none of the data in the cluster will be available. With chained declustering, data will be unavailable only if two logically adjacent drives fail – a very unlikely event. Furthermore, while Teradata's load balancing mechanism is only well suited for a certain combinations of partitioning strategies and storage organizations, the mechanisms developed for chain declustering are able to handle all combinations of partitioning strategies, storage organization, and access methods provided by Gamma.

Like the mirrored disk and Teradata strategies, chained declustering also relies on two identical copies of each relation. The technique used by RAID requires only about half as much disk space. However, since the cost of disks accounts for a relatively small fraction of the total price of a system, the performance advantage provided by chained declustering in the event of failures should more than compensate for its increased consumption of disk space, especially in the arena of high availability, high performance systems.

8.3. Future Research Directions

While we presented a new solution to the problem of high availability database systems, issues associated with the new solution also arose that merit further investigation. The first issue is how to handle skewed data access. Without data replication, data partitioning (or declustering) is commonly used with multiprocessor multi-disk database machines to break hot spots and achieve load balancing. Hot spots, however, may be dynamic in nature, and the database may need to be reorganized periodically. With chained declustering, a query (subquery) can be processed at the node storing either the primary or backup copy of the matching tuples. In addition, the work of a node may be shifted to its neighbor without physically moving data because nodes are "chained" together through the primary and backup copies of a fragment. These two characteristics of the chained declustering scheme provide a good opportunity for dynamic load balancing [Care86] when a hot spot changes over time. Consequently, a reorganization of the database may not be required. Currently, for simplicity, we assume a primary copy access scheme for chained declustering. The pros and cons of balancing the load dynamically will be an interesting area for future study.

A second issue is, in the event of a failure, whether other load balancing strategies can perform better than the load balancing algorithms described in this dissertation. In this dissertation, we presented three load balancing algorithms based on the concept of a responsible range, the use of query modification techniques, and the availability of an extent map. One other load balancing strategy that may perform equally well (or better) is to dynamically direct a subquery to the primary or backup copy node based on the current workload of the two nodes. This scheme will be especially attractive if a dynamic load balancing strategy has already been employed in the normal mode of operation. In that case, there will be no overhead of switching from normal mode to failure mode and vice versa (i.e., there is no need to recompute responsible ranges or update the system catalog where responsible range information is stored). In addition, there will be no need to implement a special software component to handle operations in the failure mode.

A third issue is to design or modify concurrency control and recovery protocols to best work with the chained declustering scheme. For example, if the dynamic load balancing strategy described above is used, a new concurrency control protocol may be needed. In addition, with multiple copies of a data item being stored, there may be room for improvement in the log based recovery protocol. Finally, we are planning to implement the chained declustering data replication scheme on the Gamma database machine or another multiprocessor multi-disk database machine.

APPENDIX

We show here how to derive p, the probability of two disks being unavailable at the same time (due to disk failures). Assuming that disk failures are independent and that the time between failures is exponentially distributed with mean MTTF, the probability that a second disk failure occurs within the repair time (assuming that the repair time is also exponentially distributed) of the first failure is [Triv82]

$$p = 1 - e^{-\frac{MTTR}{MTTF}}$$

Here, MTTR is the mean time to repair a failed disk. With current disk technology, the mean time to failure for a disk drive is between 3 and 5 years. If we assume that the mean time to failure (MTTF) of a disk is 3 years and that the time to repair/replace a disk and restore its contents (MTTR) is 5 hours, then for a pair of disks, the probability, p, of both disks being unavailable at the same time is

$$\begin{aligned} p &= 1 - e^{-\frac{5}{3\times365\times24}} \\ &= 1 - e^{-1.9\times10^{-4}} \\ &= 1 - 0.9998 \\ &= 0.0002 \end{aligned}$$

Table A.1 illustrates the probability of two disks being unavailable at the same time (PTDU) under different MTTF and MTTR assumptions.

Mean time to failure of a disk (MTTF)	Disk repair time (MTTR)	PTDU
5 years	3 hours	0.00006
5 years	5 hours	0.00011
3 years	3 hours	0.00011
3 years	5 hours	0.0002

Table A.1

REFERENCES

[Anon85] Anon et. al, "A Measure of Transaction Processing Power," TR# 85.1, Tandem Computer, Cupertino, CA, 1985.

[Atta84] Attar, R., Bernstein, P., Goodman, N., "Site Initialization, Recovery and Backup in a Distributed Database System," *IEEE Transactions on Software Engineering*, Vol. SE-10, No. 6, November 1984.

[Bern80] Bernstein, P. and Goodman, N., "Timestamp-Based Algorithms for Concurrency Control in Distributed Database Systems," *Proceedings of the 6th International Conference on Very Large Data Base*, October 1980.

[Bern84] Bernstein, P., Goodman, N., "An Algorithm for Concurrency Control and Recovery in Replicated Distributed Databases," *ACM Transactions on Database Systems*, Vol. 9, No. 4, December 1984.

[Bhar86] Bhargava, B., Ruan, Z., "Site Recovery in a Replicated Distributed Database System," *Proceedings of the 6th IEEE Distributed Computing System's Conference*, Cambridge, Mass., May 1986.

[Bitt88] Bitton, D. and J. Gray, "Disk Shadowing," *Proceedings of the 14th International Conference on Very Large Data Base*, Los Angeles, August 1988.

[Bitt89] Bitton, D., "Arm Scheduling in Shadowed Disks," *COMPCON*, IEEE Press, March 1989.

[Borr81] Borr, A., "Transaction Monitoring in Encompass [TM]: Reliable Distributed Transaction Processing," *Proceedings of the 7th International Conference on Very Large Data Base*, 1981.

[Care86] Carey, M. and H. Lu, "Load Balancing in a Locally Distributed Database System," *Proceedings of the ACM-SIGMOD International Conference on Management of Data*, 1986.

[Care89] Carey, M., Livny, M., "Parallelism and Concurrency Control Performance in Distributed Database Machines," *Proceedings of the ACM-SIGMOD International Conference on Management of Data*, Portland, Oregon June 1989.

[Ceri82] Ceri, S. and Owicki, S., "On the Use of Optimistic Methods for Concurrency Control in Distributed Databases," *Proceedings of the 6th Berkeley Workshop on Distributed Data Management and Computer Networks*, February 1982.

[Chen90] Chen, P., Gibson, G., Katz, R., and Patterson D., "An Evaluation of Redundant Arrays of Disks Using an Amdahl 5890," *Proceedings of ACM SIGMETRICS conference*, Colorado, May 1990.

[Cope88] Copeland, G., Alexander, W., Boughter, E., and T. Keller, "Data Placement in Bubba," *Proceedings of the ACM-SIGMOD International Conference on Management of Data*, Chicago, May 1988.

[Cope89] Copeland, G. and T. Keller, "A Comparison of High-Availability Media Recovery Techniques," *Proceedings of the ACM-SIGMOD International Conference on Management of Data*, Portland, Oregon June 1989.

[DeWi84] DeWitt, D., Katz, R., Olken, F, Shapiro, D., Stonebraker, M., and Wood, D., "Implementation Techniques for Main Memory Database Systems," *Proceedings of the ACM-SIGMOD Conference*, Boston, MA, June 1984.

[DeWi86] DeWitt, D., Gerber, R., Graefe, G., Heytens, M., Kumar, K., and M. Muralikrishna, "GAMMA - A High Performance Dataflow Database Machine," *Proceedings of the 12th International Conference on Very Large Data Base*, Japan, August 1986.

[DeWi88] DeWitt, D., Ghandeharizadeh, S., and Schneider, D., "A Performance Analysis of the Gamma Database Machine," *IEEE Transactions on Knowledge and Data Engineering*, Vol. 2, No 1, March 1990.

[DeWi90] DeWitt, D., Ghandeharizadeh, S.,Schneider, D., Bricker, A., Hsiao, H, and Rasmussen, R, "The Gamma Database Machine Project," *Proceedings of the ACM-SIGMOD International Conference on Management of Data*, Chicago, May 1988.

[Eckh78] Eckhouse, R. Jr., Stankovic, J., "Issues in Distributed Processing - An Overview of Two Workshops," *Computer*, January 1978.

[Eage86] Eager, D., Lazowska, E., and J. Zahorjan, "Adaptive Load Sharing in Homogeneous Distributed Systems," *IEEE Transactions on Software Engineering*, Vol. SE-12, No. 5, May 1986.

[Engl89] Englert, S., Gray, J., Kocher, T., and Shah, P., "A Benchmark of NonStop SQL Release 2 Demonstrating Near-linear Speedup and Scaleup on a Large Database" Tandem Computers, Tech. Rep. 89.4, Tandem Part No. 27469, May 1989.

[Fran87] Frank, P., "Advances in Head Technology," Presentation at *Challenges in Disk Technology Short Course*, Institute for Information Storage Technology, Santa Clara University, Santa Clara, CA, December 15-17, 1987.

[Gerb87] Gerber, R. and D. DeWitt, "The Impact of Hardware and Software Alternatives on the Performance of the Gamma Database Machine", Computer Sciences Technical Report #708, University of Wisconsin-Madison, July, 1987.

[Gibs89] Gibson, G, Hellerstein, L., Karp, R., Katz, R., and Patterson, D., "Failure Correction Techniques for Large Disk Arrays" *Proceedings of ASPLOS III*, Boston, MA., March 1989.

[Gray78] Gray, J., "Notes on Database Operating Systems," *In Operating Systems: An Advanced Course*, Vol. 60, *Lecture Notes in Computer Science*, Springer-Verlag, New York, 1978.

[Gray88] Gray, J., Sammer, H., and Whitford, S., "Shortest Seek vs Shortest Service Time Scheduling of Mirrored Disc Reads" Tandem Computers, December 1988.

[Gray90] Gray, G., Horst, B., and Walker, M., "Parity Striping of Disc Arrays: Low-Cost Reliable Storage with Acceptable Throughput," *Proceedings of the 16th Onternational Conference on Very Large Data Base*, Brisbane, Australia, August 1990.

[Hamm80] Hammer, M. and Shipman, D., "Reliability Mechanism for SDD-1: A System for Distributed Databases," *ACM Transactions on Database Systems*, Vol. 5, No. 4, December 1980.

[Jone83] Jones, S., "The Synapse Approach to High System and Database Availability," *Database Engineering*, Vol. 6, No. 2, June 1983.

[Joy85] Joy, B., Presentation at ISSCC '85 panel session, February 1985.

[Kast83] Kastner, P.C., "A Fault-Tolerant Transaction Processing Environment," *Database Engineering*, Vol. 6, No. 2, June 1983.

[Katz78] Katzman, J., "A Fault-Tolerant Computing System," *Proceedings of the 11th Hawaii Conference on System Sciences*, January 1978.

[Kim86] Kim, M., "Synchronized Disk Interleaving," *IEEE Transactions on Computers*, Vol. C-35, No. 11, November 1986.

[Kung81] Kung, H. and Robinson, J, "On Optimistic Methods for Concurrency Control," *ACM Transactions on Database Systems*, Vol. 6, No. 2, June 1981.

[Lind81] Lindsay, B., P. Selinger, C. Galtieri, J. Gray, R. Lorie, T. Price, F. Putzulo, I. Traiger, B. Wade, "Notes on Distributed Databases," *In Distributed Databases, Drattan and Poole, Eds.*, Cambridge University Press, New York, 1981.

[Livn82] Livny, M., M. Melman, "Load Balancing in Homogeneous Broadcast Distributed Systems," *Proceedings of ACM Computer Network Performance Symposium*, April 1982.

[Livn87] Livny, M., S. Khoshafian, and H. Boral, "Multi-Disk Management," *Proceedings of ACM SIGMETRICS Conference*, Alberta, Canada, 1987.

[Livn89] Livny, M., "DeNet User's Guide," Version 1.5, Computer Sciences Department, University of Wisconsin-Madison, 1989.

[Mena78] Menasce, D. and Muntz, R., "Locking and Deadlock Detection in Distributed Databases," *Proceedings of the 3rd Berkeley Workshop on Distributed Data Management and Computer Networks*, August 1978.

[Moha89] Mohan, C., Haderle, D., Lindsay, B., Pirahesh, H., and Schwarz, P., "ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging", RJ 6649, IBM Almaden Research Center, San Jose, California, January 1989.

[Patt88] Patterson, D., Gibson, G., and Katz, R., "A Case for Redundant Arrays of Inexpensive Disks (RAID)," *Proceedings of the ACM-SIGMOD International Conference on Management of Data*, Chicago, May

1988.

[Reed83] Reed, D., "Implementing Atomic Actions on Decentralized Data," *ACM Transactions on Computer Systems*, Vol. 1, No. 1, February 1983.

[Ries78] Ries, D. and Epstein, R., "Evaluation of Distribution Criteria for Distributed Database Systems," UCB/ERL Technical Report M78/22, UC Berkeley, May 1978.

[Sale84] Salem, K. and H. Garcia-Molina, "Disk Striping," EECS TR# 332, Princeton University, Princeton, NJ, December 1984.

[Schl81] Schlageter, G, "Optimistic Methods for Concurrency Control in Distributed Database Systems," *Proceedings of the 7th International Conference on Very Large Data Bases*, Cannes, France, September 1981.

[Schu88] Schulze, M., Gibson, G., Katz, R., and Patterson, D., "How Reliable Is A RAID?" Unpublished Technical Report, 1988.

[Ston75] Stonebraker, M., "Implementation of Integrity Constraints and Views by Query Modification," *Proceedings of the SIGMOD Workshop on Management of Data*, San Jose, Calif., May 1975.

[Ston86] Stonebraker, M., "The Case for Shared Nothing," *Database Engineering*, Vol. 9, No. 1, 1986.

[Tand87] Tandem Database Group, "NonStop SQL, A Distributed, High-Performance, High-Reliability Implementation of SQL," *Workshop on High Performance Transaction Systems*, Asilomar, CA, September 1987.

[Tand88] The Tandem Performance Group, "A Benchmark of NonStop SQL on the Debit Credit Transaction," *Proceedings of the ACM-SIGMOD International Conference on Management of Data*, Chicago, May 1988.

[Teor72] Teorey, T., and Pinkerton, T., "A Comparative Analysis of Disk Scheduling Policies," *Communications of ACM*, Vol. 15, No. 3, March 1972.

[Tera85] Teradata, "DBC/1012 Database Computer System Manual Release 2.0," Document No. C10-0001-02, Teradata Corp., NOV 1985.

[Triv82] Trivedi, K., in *Probability and Statistics with Reliability, Queueing, and Computer Science Applications*, Prentice-Hall Inc., New Jersey, 1982.

[Thom79] Thomas, R., "A Majority Consensus Approach to Concurrency Control for Multiple Copy Databases," *ACM Transactions on Database Systems*, Vol. 4, No. 2, June 1979.