Chained Declustering: A New Availability Strategy
for
Multiprocssor Database machines

Hui-I Hsiao
David J. DeWitt

Computer Sciences Department
University of Wisconsin

This research was partially supported by the Defense Advanced Research Projects Agency under contract N00039-86-C-0578, by the National Science Foundation under grants DCR-8512862, MCS82-01870, and MCS81-05904, and by a Digital Equipment Corporation External Research Grant.

Abstract

This paper presents a new strategy for increasing the availability of data in multi-processor, 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.

1. Introduction

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 software modules can continue running the 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. One approach for obtaining high availability is to replicate data items on separate disks attached to separate processors [Borr 81, Tera85]. 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. An alternative approach is to use an array of disks and store the data and the redundant error detection and correction information (usually, parity bytes) on different disk drives [Kim86, Patt88]. When errors are discovered, the redundant information can be used to restore the data and the application program can continue using the data with minimum interruption. The advantage of the first approach is higher availability and better performance for transaction-oriented database applications. The merit of the second approach is that it uses less disk space (i.e., less redundant information is stored) and potentially provides higher I/O bandwidth for large block transfers. While the identical copy scheme (which is employed by both Tandem and Teradata in their systems) has been used commercially, the suitability of the second approach for database applications is still under investigation.

Throughout this paper, 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.

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[1], and the **failure mode**, in which one or more processors or disks have failed. In the normal mode of operation, the application of both intra and inter query parallelism has proven successful at improving the performance of database management system software [Tera85, 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[2] (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 as a bottleneck may form.

For multiprocessor, shared-nothing database machines with replicated data, the application of horizontal partitioning (ie. declustering) techniques [Ries78, Tera85, DeWi86, Livn87] facilitates the successful application of inter and intra query parallelism in the normal mode of operation. None of the existing availability techniques, however, are able to fully[3] balance the workload when failures occur. In this paper we present a new declustering technique, termed **chained declustering**, for managing replicated data. This new technique is capable of providing both high availability in addition to being able to fully balance the workload among the operational nodes in the event of a failure. To balance the workload in the failure mode, a "static" 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 reassigned when node failures occur.

In the next section, related research dealing with replication is presented. The chained declustering strategy is described in Section 3 along with the associated data placement algorithms. The operation of the static load balancing scheme is described in Section 4. Section 5 contains a comparison of the availability and performance of the

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

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

[3] 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.

chained declustering strategy with existing availability techniques. Our conclusions and future research directions are contained in Section 6.

2. Related Availability Strategies

In this section, we briefly describe several existing techniques for improving data availability including the use of mirrored disks [Borr81], data clustering [Tera85], and disk arrays with redundant check information [Patt88]. The mirrored disk and declustering strategies employed, respectively, by Tandem and Teradata, both maintain two identical copies of each relation. Bubba [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 [Kim86] 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 sections below, we will describe Tandem's mirrored disk scheme, Teradata's data clustering scheme, and RAID's data placement scheme in additional detail.

2.1. 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 [REF] system are generally declustered across multiple disk drives. For example, Figure 1 shows relation R partitioned across four disks using the mirrored disk strategy. $R_P(i)$ represents the i-th horizontal fragment of the first copy of R and $R_B(i)$ stands for the mirror image of $R_P(i)$. As shown in Figure 1, the contents of disks 1 and 2 (and 3 and 4) are identical.

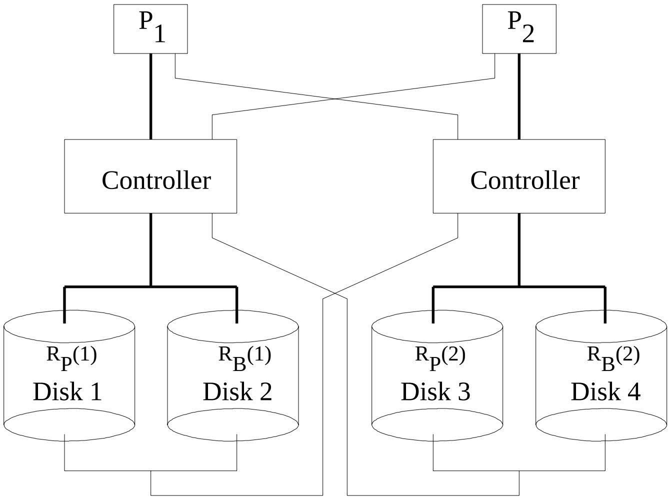

Figure 1: Data Placement with Tandem's Mirrored Disk Scheme.

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 mirror-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 both drives in order to keep the contents of both disks identical, causing the two disk arms to become synchronized [REF].

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 [REF]. 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 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.

2.2. Teradata's Data Clustering 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.[4] Relations are declustered among the disk drives within one or more clusters by hashing on a "key" attribute. (The tuples of a relation stored on each disk are termed a **fragment**.) 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. 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. 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. When a processor 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 processor will be distributed among N-1 processors instead of a single processor. However, this improvement in load balancing is not without cost. As we will demonstrate in Section 5, the probability of data being unavailable increases proportionately with the size of the cluster.[5]

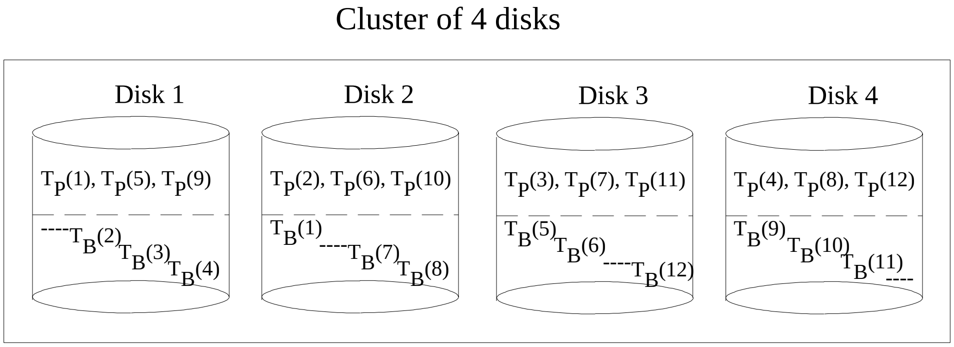

Figure 2: Teradata's Data Clustering Scheme.

Figure 2 illustrates Teradata's data replication scheme. Relation R is declustered over a cluster consisting of four processors and disks by hashing on the "key" attribute to produce hash values ranging from 1 to 12. As shown in Figure 2, tuples from the primary copy which hash to values 1, 5, and 9 are placed on disk 1 while tuples from the backup copy having have the same hash values (1, 5, and 9) are stored on disks 2, 3 and 4, respectively. $T_P(i)$

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

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

stands for all tuples from the primary copy whose partitioning attribute value hash to value *i* and $T_B(i)$ is the backup copy of $T_P(i)$.

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.

2.3. 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 3 illustrates an example of data placement in a RAID[6] 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 3, the check byte sectors in a RAID group are distributed evenly among the disk drives.

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 proceeds as follows. 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. If the two bits match, the failed bit is a 0. Otherwise, it is a 1.

[6] [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].

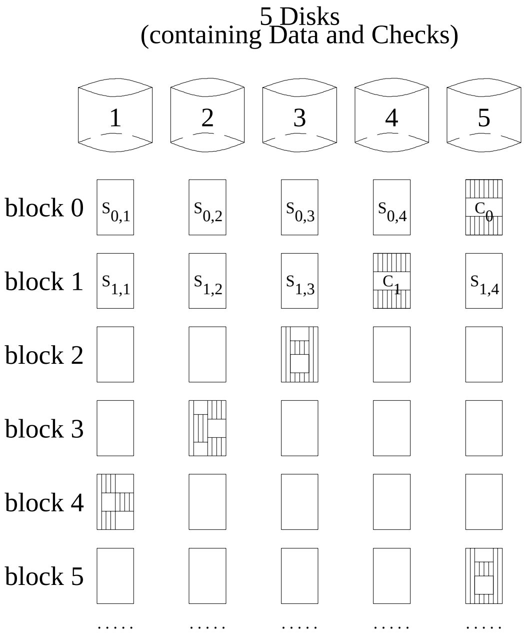

Figure 3: Data placement in RAID, each rectangle represents a physical sector.

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 unavailable 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.

3. Chained Declustering Strategy

While each of the techniques presented in the previous section 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 Section 5), it does a poor job of distributing the load of a failed processor. 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. RAID emphasizes the importance of disk space over performance.

In this section, 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 RAID strategy. 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.

3.1. Data Placement Algorithm for Chained Declustering

In the chained declustering strategy, nodes are divided into disjoint groups called *relation-clusters*. The tuples of each relation in the database are declustered across the disk drives of one relation cluster. The chained declustering strategy maintains two physical copies of each relation. The first copy (termed the *primary copy*) is declustered over all disks in a *relation-cluster* using one of Gamma's three partitioning strategies (hashed, range, or round-robin) [DeWi86]. The second copy (termed the *backup copy*) is declustered using the same partitioning strategy but the corresponding fragments from both copies are stored on different nodes. The fragments are distributed among the disks using the following algorithm:

> Assume that there are a total of M disks numbered from 1 to M. For every relation R, the i-th fragment of the primary copy is stored on the {[i-1+C(R)] mod M + 1}-th disk drive, and the i-th fragment of the backup copy will be stored on the {[i+C(R)] mod M + 1}-th[7] disk drive; insuring that the primary and backup fragments are placed on different disk drives.

The function C(R) is devised to allow the first fragment of relation R to be placed on any disk within a *relation-cluster*. We name this data replication method **chained declustering** because the disks are linked together, by the fragments of a relation, like a chain.

Figure 4 illustrates the data placement scheme with chained declustering where the size of the *relation-cluster* is eight and C(R) is 0. $R_P(i)$ and $R_B(i)$ stand for the i-th fragment of the primary and backup copy, respectively. In the figure, $R_P(1)$ and $R_B(1)$ contain identical data, and so on.

For systems having relatively few disks, it is reasonable to let the *relation-cluster* contain all 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 the drives as the overhead of initiating and committing transactions will generally overshadow the benefits obtained through the use of parallelism [Smith Sigmod 88 REF]. In this situation, the set of disks can be

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

Data placement in chained declustering scheme								
Node #	1	2	3	4	5	6	7	8
Primary copy of R	$R_P(1)$	$R_P(2)$	$R_P(3)$	$R_P(4)$	$R_P(5)$	$R_P(6)$	$R_P(7)$	$R_P(8)$
Backup copy of R	$R_B(8)$	$R_B(1)$	$R_B(2)$	$R_B(3)$	$R_B(4)$	$R_B(5)$	$R_B(6)$	$R_B(7)$

Figure 4: Replicated Data Placement in the Chained Declustering Scheme.

divided into groups and each group will form a **relation-cluster**. The data placement algorithm described previously can be modified to adapt to this change by adding a term to represent the starting disk number of the relation-cluster. In addition, when the number of disks in a relation-cluster is relatively large (e.g. 100-200 disks), the disks in a *relation-cluster* can be divided into even smaller groups termed **chain-clusters**, which are then the unit of load balancing in the event of a node failure. The data placement algorithm described earlier can be further modified to accommodate this grouping mechanism.

In summary, the chained declustering algorithm declusters a relation over any M disks out of a total of L disks (2 <= M <= L), and "chains" groups of N disks (2 <= N <= M). For special cases of 2 <= N <= 16, chained declustering is similar to, but, as will be demonstrated later, provides higher data availability than Teradata’s cluster scheme. Figure 5 illustrates data placement for the chained declustering scheme with N = 4 and L = M = 8. As shown in the figure, the first four fragments are "chained" within the first four disks and the last four fragments are "chained" in disks 5 through 8.

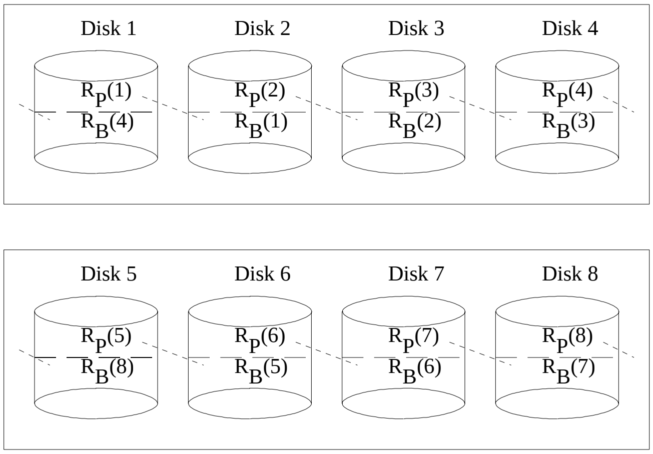

Figure 5: $R_P(i)$ and $R_B(i)$ are the i-th fragments of the primary and backup copy, respectively.

3.2. Availability and Load Balancing

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 failure rates are exponentially distributed[8] with a mean 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 = 2), the probability, *p*, of both disks failing at the same time is about 0.0002. For M = 8, M = 32, M = 250, and M = 1000, the probability of two disks failing at the same time will be 0.0014, 0.0062, 0.046, and 0.17, 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[9] when M = 32, once every 3.2 months when M = 250, and once every 6.4 days when M = 1000. With chained declustering, however, 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 0.0004 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 6.4 days.

The unique capability of the chained declustering scheme is that it is able to balance the workload among all disk drives during both the normal and failure modes of operation. Figure 6 illustrates the idea of how the workload is balanced in the event of a node (processor or disk) failure. The fragments from the primary copy are initially designated as the active fragments ($R_P(i)$ and $R_B(i)$ are the i-th fragments of the primary copy and backup copy of the relation R, respectively). From part (ii) in Figure 6, one can see that when a node fails, its workload is shifted to its backup node as in the case of $R_P(2)$. Consequently, the active fragments for relation R are dynamically reassigned such that the workload is uniformly distributed among all remaining nodes. As illustrated by part (ii) of Figure 6, the workload of node 3 is increased only by 14.3%. If the active fragments had not been rearranged, the workload of node 3 would have instead increased by 100%[10]. As shown in parts (ii) and (iii) of Figure 6, the fraction of the primary and backup fragments that a node is "responsible for" varies depending on exactly which node has failed.

[8] The exponential distribution has an important property, known as the **memoryless property**.

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

[10] In this case, node 3 would be responsible for both $R_P(3)$ and $R_B(2)$ and its work load will be increased by 100%. This analysis assumes, of course, that the workload was uniformly distributed among the nodes before the failure occurred.

Node #	1	2	3	4	5	6	7	8
Primary fragments of R	$R_P(1)$	$R_P(2)$	$R_P(3)$	$R_P(4)$	$R_P(5)$	$R_P(6)$	$R_P(7)$	$R_P(8)$
Backup fragments of R	$R_B(8)$	$R_B(1)$	$R_B(2)$	$R_B(3)$	$R_B(4)$	$R_B(5)$	$R_B(6)$	$R_B(7)$

(i) Relation R distributed with Chained Declustering scheme.
The Primary Fragments are the Active Fragments.

Node #	1	2	3	4	5	6	7	8
New active fragments of relation R	$R_P(1)$ + $\frac{1}{7}R_B(8)$	---- - ----	$\frac{1}{7}R_P(3)$ + $R_B(2)$	$\frac{2}{7}R_P(4)$ + $\frac{6}{7}R_B(3)$	$\frac{3}{7}R_P(5)$ + $\frac{5}{7}R_B(4)$	$\frac{4}{7}R_P(6)$ + $\frac{4}{7}R_B(5)$	$\frac{5}{7}R_P(7)$ + $\frac{3}{7}R_B(6)$	$\frac{6}{7}R_P(8)$ + $\frac{2}{7}R_B(7)$

(ii) Reassignment of Active Fragments after Node 2 Fails[10].

Node #	1	2	3	4	5	6	7	8
New active fragments of relation R	$\frac{4}{7}R_P(1)$ + $\frac{4}{7}R_B(8)$	$\frac{5}{7}R_P(2)$ - $\frac{3}{7}R_B(1)$	$\frac{6}{7}R_P(3)$ + $\frac{2}{7}R_B(2)$	$R_P(4)$ + $\frac{1}{7}R_B(3)$	--- + ---	$\frac{1}{7}R_P(6)$ + $R_B(5)$	$\frac{2}{7}R_P(7)$ + $\frac{6}{7}R_B(6)$	$\frac{3}{7}R_P(8)$ + $\frac{5}{7}R_B(7)$

(iii) Reassignment of Active Fragments after Node 5 Fails.
Figure 6: Assignment of the Active Fragments before and after a Node Failure.

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 this process can be done very quickly and efficiently. If queries do not exhibit a uniform access pattern, the chained declustering scheme also allows dynamic load balancing without actually moving the physically stored data. Since the data fragments are "chained" together, the work of one node can be easily reassigned to either of its neighbors. The technique of dynamic load balancing is currently under investigation and will not be discussed further in this paper.

4. Load Balancing Algorithms for Chained Declustering

The example shown in Figure 6 provides a simplified view of how the chained declustering mechanism actually balances the workload in the event of a node failure. In reality, 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 optimizers 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,

[10] The $\frac{n}{7}$ of a fragment ($R_P(i)$ or $R_B(i)$) for a node is
the **fraction of responsibility** of the i-th fragment at that node.

Gamma also supports five storage organizations for each relation[11]:

(1) a clustered index on the partitioning attribute,

(2) a clustered index on the 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 tuples. There are: (1) utilize the clustered index, (2) utilize a non-clustered index, and (3) a sequential file scan.

The problem now is to design a load balancing algorithm for the chained declustering mechanism that handles 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.

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

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

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

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

M is the total number of nodes in the *relation-cluster*. $W_l(x)$ and $W_h(x)$ correspond to the lower and upper bound values of attribute W for fragment x. For example, if M=4 and S=2, then $f(3,S)= \frac{2}{3}$. Assuming that $W_l(r_P)$ is 61

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

and $W_h(r_P)$ is 90, when node 2 fails the responsible range of node 3 on attribute W for its primary fragment r_P is $\left[61,\ \ 90 - \frac{2}{3} * 30\right] = \left[61,\ \ 70\right]$.

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 i for a fragment is determined by the following formulas:

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

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

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

Figure 7 illustrates the structure of an extent map. As shown in Figure 7, the extent map for a fragment (x) 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 j-th extent will consist of sectors $(j\text{-}1)*\frac{T}{(M-1)}+1$ to $j*\frac{T}{(M-1)}$, where T is the total number of sectors occupied by fragment x at node A. Figure 8 illustrates the responsible extents of each node before and after a node failure for a relation that is 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

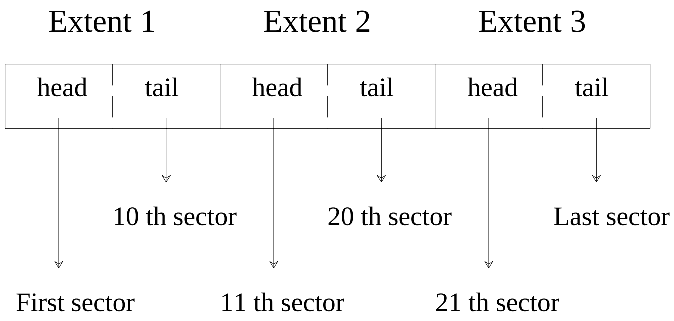

Figure 7: An example of an extent map with M=4 and T=30.

Active fragment table

Node #	1		2		3		4	
Fragment	1_P	4_B	2_P	1_B	3_P	2_B	4_P	3_B
Responsible extents (no failure)	extents 1 - 3	---	extents 1 - 3	---	extents 1 - 3	---	extents 1 - 3	---
Responsible extents (node 2 failed)	extents 1 - 3	extent 3	failed	---	extent 1	extents 1 - 3	extents 1 - 2	extents 2 - 3

Figure 8: Active Fragment Table for a Relation Stored as a Heap.

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[12] will be increased by $\frac{1}{M-1}$th. This, in turn, will increase the workload of all active nodes within the relation-cluster by $\frac{1}{M-1}$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.

4.2. Load Balancing Algorithm

In order to insure 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

[12] Or the chain-cluster, if the relation cluster has been divided into two or more chain-clusters.

are accessed. In this situation, the extent map described in Section 4.1 will be used.

Algorithm A: Changing the Selection Predicate

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

> The query scheduler first computes the **intersection** of the range of the selection predicate on W and the responsible range of node A on W.. If the **intersection** of the two ranges is non-empty, a 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 local 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 $X_l(i)$ to $X_h(i)$). A selection predicate on attribute X corresponding to the current responsible X range of fragment i (i.e. $X_l(i) \leq R.X \leq X_h(i)$) is appended to the original query before the query is sent to node A. After receiving a query, node A retrieves tuples sequentially from $X_l(i)$ to $X_h(i)$ because the tuples in its fragment are sorted on their X attribute values.

Algorithm C: Using the Extent Map

When relations are 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 4.1 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 both the primary and backup fragments. Therefore, extents from both fragments will be processed. The retrieval process for a node is completed when all tuples within its responsible range have been accessed.

4.3. Examples Applications of the Load Balancing Algorithms

To illustrate the operation of the load balancing algorithm, 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 120 and 1 to 30, respectively. We further assume, without the loss of generality, that there are four nodes in the system, that the *relation-cluster* and the *chain-cluster* both consist of four nodes (M=4), and that the second node (S=2) has failed.

In the following examples, a B subscript (e.g. R_B) is used to indicate that the fragment from the backup copy of relation R must be processed. A range variable without a subscript (e.g. R.Z) indicates that the primary fragment of the relation is to be used. The examples show how the load balancing algorithms described in the previous section are used when node 2 fails.

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. The primary and backup fragments of relation R at a local node will have disjoint sets of X attribute values. Figure 9 illustrates the assignment of responsible ranges (active fragments) for attribute X before and after the failure of node 2.

Active fragment table								
Node #	1		2		3		4	
Fragment	1_P	4_B	2_P	1_B	3_P	2_B	4_P	3_B
Responsible X ranges	[1,30]	---	[31,60]	---	[61,90]	---	[91,120]	---
Backup ranges	---	[91,120]	---	[1,30]	---	[31,60]	---	[61,90]

i: Before any node failure.

Active fragment table							
Node #	1		2	3		4	
Fragment	1_P	4_B	2_P 1_B	3_P	2_B	4_P	3_B
Responsible X range	[1,30]	[111,120]	failed	[61,70]	[31,60]	[91,110]	[71,90]

ii: After the failure of node 2.

Figure 9: An example of an active fragment table for attribute X.

For exact match selection queries on attribute X (e.g. R.X = constant), the access plan will specify that the clustered index is to be used to access relation R. At run-time, the query scheduler will use the active fragment table to direct the query to the appropriate node(s) for execution. For example, given the active fragment (responsible range) assignment indicated in Figure 9, queries with predicates R.X = 43, R.X = 81, and R.X = 115 will be directed respectively to nodes 2, 3, and 4 in the normal mode of operation and will be directed to nodes 3, 4, and 1, respectively, in the event that node 2 has failed.

In the case of range queries on attribute X (e.g. c1 ≤ R.X ≤ c2), when a node has failed, the query scheduler will modify the ranges of each query before sending the query to the nodes where the corresponding active fragments are stored (ie., Algorithm A in Section 4.2 is employed). Figure 10 illustrates how a query is modified[13] after the failure of node 2. In Figure 10, the range of the query is from 50 to 80. The responsible X range of nodes 3 and 4 are from 31 to 70 and from 71 to 110 (from Figure 9), respectively. The intersection of "50 to 80" (the query range) and "31 to 70" (node 3's responsible range) is "50 to 70", so the original query is modified to "R.X ≥ 50 **and** R.X < 70"[14] before it is sent to node 3. The query modification process for node 4 is done in a similar fashion. For node 1, the *intersection* of the query range and the responsible X range is empty and the query (neither the original nor the modified version of it) will not be sent to it. At nodes 3 and 4, the modified query will be applied to the active fragments stored at each site.

retrieve ....
where R.X ≥ 50 **and** R.X < 80

i: Original query

retrieve ....
where R.X ≥ 50 **and** R.X ≤ 70

iii: Modified query for node 3

ii: node 1 is not responsible for the query

retrieve ....
where R.X ≥ 71 **and** R.X < 80

iv: Modified query for node 4

Figure 10: An Example of Query Modification.

Example #2: Non-clustered index on non-partitioning attribute is used to retrieve the qualified tuples.

[13] To simplify the explanation, the example uses the original query. In an actual implementation, the algorithm will modify the query execution plan which is generated by the query optimizer.

[14] This is a special case. The R.X here means R.X and/or R_B.X. Because the two fragments from the same relation are disjoint and ordered at a local node, the index(clustered) trees for these two fragments can be joined together and form a single tree.

Assume that a non-clustered index is constructed on attribute Z and that attribute Z has values ranging from 1 to 30. Figure 11 depicts the responsible range of the remaining three nodes after node 2 has failed. Because each of the fragments have the same range of Z values, after the reassignment, the responsible range of each of these 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 site 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 4.2 to modify the query before it is sent to the appropriate nodes for processing. Figure 12 illustrates how a range query is modified after the failure of node 2. The query is sent unmodified to node 1 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 1's responsible Z range for the backup fragment is empty. For node 4, 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 11 to 14). As a result, the query is modified to include both intersections before it is sent to node 4. The query modification for node 3 can be done in a similar fashion and is not shown here. The subscript in R_B serves to inform a node that the fragment from the backup copy of relation R must be processed.

Active fragment table								
Node #	1		2		3		4	
Fragment	1_P	4_B	2_P	1_B	3_P	2_B	4_P	3_B
Responsible Z ranges	[1,30]	---	[1,30]	---	[1,30]	---	[1,30]	---
Backup ranges	---	[1,30]	---	[1,30]	---	[1,30]	---	[1,30]

i: Before any node failure.

Active fragment table								
Node #	1		2		3		4	
Fragment	1_P	4_B	2_P	1_B	3_P	2_B	4_P	3_B
Responsible Z ranges	[1,30]	[21,30]	failed		[1,10]	[1,30]	[1,20]	[11,30]

ii: After the failure of node 2.

Figure 11: An example of active fragment table for indexed attribute Z (relation not partitioned on Z).

retrieve ....
where $R.Z \geq 6$ **and** $R.Z \leq 14$

i: Original query

retrieve ....
where $R.Z \geq 6$ **and** $R.Z \leq 14$

ii: Original query for node 1

retrieve ....
where $R.Z \geq 6$ **and** $R.Z \leq 10$
or $R_B.Z \geq 6$ **and** $R_B.Z \leq 14$

iii: Modified query for node 3

retrieve ....
where $R.Z \geq 6$ **and** $R.Z \leq 14$
or $R_B.Z \geq 11$ **and** $R_B.Z \leq 14$

iv: Modified query for node 4

Figure 12: Query modification for range queries on non-partitioning attribute.

Example #3: 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, r(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} ($q_{max} = \frac{2^{32}-1}{M}$) 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 4.1 (when recomputing the responsible range on h(X), 0 is substituted for W_l and q_{max} is substituted for W_h). Figure 13 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 120. For purposes of this example, assume that the hash function h generates only a 16-bit hash value (thus, $q_{max} = \frac{2^{16}-1}{4} = 16383$). For simplicity we use "0 to 120" 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 "0 to 120".

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 r(X). r(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 $\frac{30770}{4}$ = 7692 and r(X=97) will be equal to (30770 **mod** 4) = 2. r(X)= 2 indicates that the qualified tuple(s) is in the third fragment (tuples hashed to r(X)= i are stored in the (i+1)-th fragment). Because the third fragment is stored at two nodes (the primary copy is stored at node 3 and the backup copy is stored at node 4), q(X) is used to select a node by comparing the responsible h(X) ranges of these two nodes for the third fragment. According to Figure 13 (part ii), q(X) is within node 4's responsible h(X) range for the third fragment ("5461 to q_{max}" for 3_B), so the query "R.X = 97"[15] will be sent to node 4 for processing. Upon receiving a query, a local node will use the clustered index to retrieve qualified tuple(s).

Active fragment table

Node #	1		2		3		4	
Fragment	1_P	4_B	2_P	1_B	3_P	2_B	4_P	3_B
Responsible X ranges	[1,120]	---	[1,120]	---	[1,120]	---	[1,120]	---
Responsible h(X) ranges	$[0,q_m]$	---	$[0,q_m]$	---	$[0,q_m]$	---	$[0,q_m]$	---

i: Before any node failure. ($q_m = \frac{2^{16}-1}{M}$ = 16383)

Active fragment table

Node #	1		2	3		4	
Fragment	1_P	4_B	2_P 1_B	3_P	2_B	4_P	3_B
Responsible X ranges	[1,120]	[81,120]	failed	[1,40]	[1,120]	[1,80]	[41,120]
Responsible h(X) ranges	$[0,q_m]$	$[10922,q_m]$	failed	[0,5460]	$[0,q_m]$	[0,10921]	$[5461,q_m]$

ii: After the failure of node 2.

Figure 13: Assignment of Responsible Ranges for Hash Partitioning Attribute X.

[15] Since the third fragment stored at node 4 is from the backup copy, the query will be modified to "R_B.X = 97" before it is sent to node 4.

4.4. 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 1 summarizes the operation of the load balancing algorithm for all possible combinations of partitioning methods, storage organizations, and access plans.

RANGE PARTITIONING	Clustered on X	Clustered on Y	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 1: Summary of Load Balancing Algorithm (X is the partitioning attribute.)

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 queries to all active nodes within the relation-cluster

f: send queries to limited nodes

g: select nodes based on X value if X is the qualified attribute

h: select node based on responsible h(X) range for exact match queries

N/A: not applicable

5. Availability and Performance

While availability can be improved through the use of data replication and performance can be improved through the use of declustering and parallelism, it is not a straightforward task to simultaneously achieve both objectives in the event of processor or disk failures. The tension between these two objectives is illustrated by Figures 14 and 15 (which have been extracted from [Tera85]). Figure 14 illustrates that, for a system containing 32 nodes, as the cluster size is increased from 2 to 32 processors the mean time between cluster failures(MTTF_c) decreases dramatically. In other words, an increase in the cluster size reduces the probability that all data will be available. On the other hand, Figure 15 demonstrates that larger cluster sizes have the beneficial effect of reducing the increase in the workload on the remaining processors when a node in a cluster fails. As we will demonstrate below, chained declustering is, however, able to resolve these conficts as the probability of data being unavailable (as the result of two nodes failing) with chained declustering is independent of the cluster size.

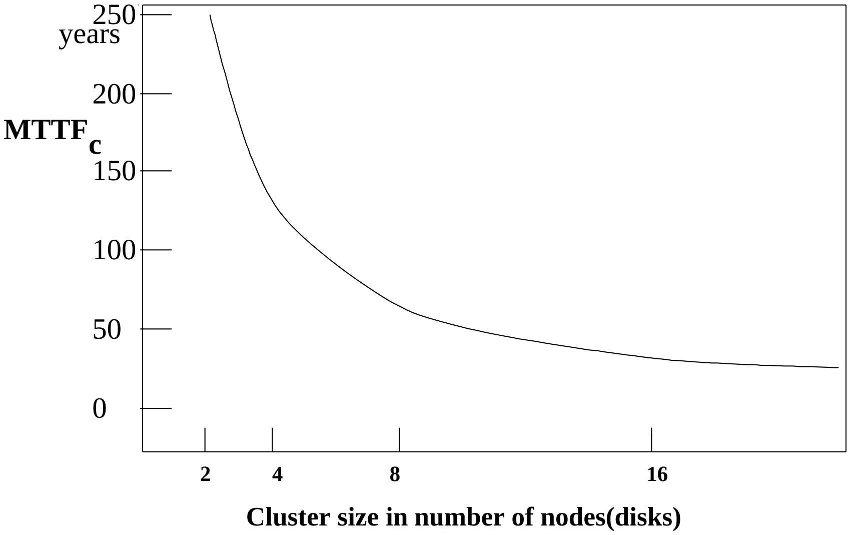

Figure 14: Mean Time Between Cluster Failures (2 failed nodes) vs. Cluster Size.

A cluster size of 32 is not shown. The MTTF_c is defined as two or more disks on different nodes in a cluster being down at the same time.

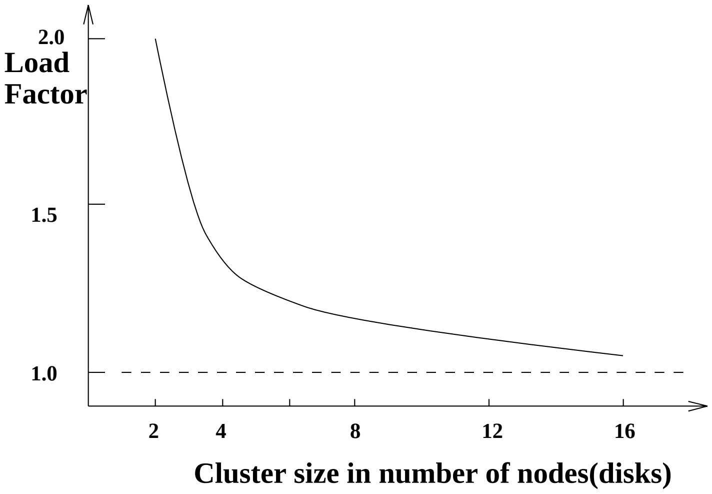

Figure 15: Excess Load on Cluster If One Node Fails vs Cluster Size.

5.1. Availability

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

- A relation is distributed across M disks,
- The size of a *chain-cluster* is also M,
- The cluster size of Teradata's 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., $n_1 = \binom{M}{1}$. n_2, however, depends on the data placement scheme used.

With Teradata's clustering scheme, if the second failure is in the same cluster as the first one, data will become unavailable. 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 1. In 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.

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 paper, their respective E values are:

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

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

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

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

Let P be the probability of any data being unavailable. Then P will be equal to E multiplied by *p* (the probability of two disks failing at the same time), i.e. P(scheme) = E(scheme) * *p*.

As one can see from the analysis above, the mirrored disk scheme provides the highest degree of data availability while the chained declustering technique is second. Both of these schemes have a constant P value when M is fixed. In contrast, the P value of Teradata's clustering scheme varies significantly depending on the cluster size, N. With Teradata's scheme, if N=8, the probability of data being unavailable will be 350% higher than with the chained declustering scheme. If N=32, it will be 1550% higher. RAID has the same availability as the Teradata 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. With G = 25, RAID will be 24 times more likely to have data unavailable than when mirrored disks are used and 12 times more likely than with chained declustering.

5.2. Load Balancing and Performance

Although the mirrored disk scheme provides the highest degree of data availability, it's major drawback is that it cannot balance the workload when a node fails. For example, assume that files A and B are stored on two mirrored disk pairs attached to processors 1 and 2. When either processor 1 or 2 fails, access to both files will be directed to the same mirrored pair and the workload on the remaining processor will double. In contrast, the chained declustering scheme can evenly distribute the workload among all remaining nodes when any node fails. To recall an example given earlier, with M = 8, when one node fails, the workload of its "backup" node will increase by only $\frac{1}{7}$.

In the event of a node failure, Teradata’s clustering scheme provides limited load balancing depending on the cluster size. In order to improve load balancing (i.e. a larger cluster size), the Teradata scheme must pay the price of a higher probability of data being unavailable. On the other hand, in order to reduce the probability of unavailable data (i.e. a smaller cluster size), load balancing in the event of a node failure will be poorer.

In the RAID data placement scheme, the system will be restricted to one disk I/O per recovery group (G disks) when one tries to access data on the failed disk before it is repaired/replaced. This is true because all remaining G-1 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 reconstructing process will compete with the "normal" processes for both CPU and I/O bandwidth. System performance, in terms of both throughput and response time, will degrad 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.

5.3. Summary

In Table 3 we have summarized the probability of data not being availabile and the impact of a processor failure on the workload of the remaining processors.

Schemes	Chained declustering	Mirrored disks	RAID’s data placement	Teradata’s clustering
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%
M = 32 and 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%
M = 32 and 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%

Table 3: Summary of Availability and Load Balancing in Different Data Placement Schemes.

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

6. Conclusions and Future Research Directions

In this paper 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, is 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 able to handle relations that are hash-partitioned, the mechanisms developed for chain declustering are able to handle all combinations of partitioning mechanisms 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 techniques 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 system expense, 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 the highly available, high performance systems.

The analysis in this paper assumes that each processor is responsible for one disk[16]. One issue that was not addressed is whether or not the results obtained will be significantly different if more than one disk is connected to a processor. With respect to availability, the answer is no. Whether data will be available or not depends on how and where it is placed and this is decided by the data placement scheme. Any other disk, whether connected to the same processor as the original or not, will not affect its availability. For load balancing, on the other hand, having multiple disks connected to one processor tends to reduce the relative increase in workload on the remaining processors in the event of a disk/processor failure. The absolute increase in workload on the remaining processors, however, is increased. If one considers the workload of a disk (disk queue length), the result will remain the same.

In the future we intend to conduct a detailed performance study of the chained declustering mechanism. Among the issues we intend to investigate are the impact of a single node failure on different query types, the overhead (query modification, maintenance of active fragment table and extent maps) associated with chained declustering, and the impact on performance of having to update both copies of each relation.

REFERENCES

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

[16] Except for RAID.

[Bitt88] Bitton, D. and J. Gray, "Disk Shadowing," *Proceedings of VLDB,* Los Angeles, August 1988.

[Borr81] Borr, A., "Transaction Monitoring in Encompass [TM]: Reliable Distributed Transaction Processing," *Proceedings of VLDB,* 1981.

[Care85] Carey, M., Livny, M., and H. Lu, "Dynamic Task Allocation in a Distributed Database System," *Proceedings of the 5th International Conference on Distributed Computer Systems,* Denver, May 1985.

[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.

[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.

[DeWi86] DeWitt, D., Gerber, R., Graefe, G., Heytens, M., Kumar, K., and M. Muralikrishna, "GAMMA-A High Performance Dataflow Database Machine," *Proceedings of the 1986 VLDB Conference,* Japan, August 1986.

[DeWit88] DeWitt, D., Ghandeharizadeh, S., and D. Schneider, "A Performance Analysis of the Gamma Database Machine," *Proceedings of the ACM-SIGMOD International Conference on Management of Data,* Chicago, May 1988.

[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.

[Gibs89] Gibson, G, Hellerstein, L., Karp, R., Katz, R., and D. Patterson, "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.

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

[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.

[Lind86] Lindsay, B., P. Seilinger, C. Galtieri, J. Gray, R. Lorie, T. Price, F. Putzulo, I. Traiger, and B. Wade, "Notes on Distributed Databases," *In Distributed Databases, Drattan and Poole, Eds.,* Cambridge University Press, New York, 1980.

[Livn82] Livny, M. and 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.

[Patt88] Patterson, D. A., G. Gibson, and R. H. Katz, "A Case for Redundant Arrays of Inexpensive Disks (RAID)," *Proceedings of the ACM-SIGMOD International Conference on Management of Data,* Chicago, May 1988.

[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.

[Schu88] Schulze, M, G. Gibson, R. Katz, and D. Patterson, "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, SEP 1987.

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