PHYSICAL DATABASE DESIGN IN MULTIPROCESSOR DATABASE SYSTEMS

by

Shahram Ghandeharizadeh

Computer Sciences Technical Report #964
September 1990

PHYSICAL DATABASE DESIGN IN MULTIPROCESSOR DATABASE SYSTEMS

by

SHAHRAM GHANDEHARIZADEH

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

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.

ABSTRACT

In shared-nothing multiprocessor database machines, relations are horizontally declustered across multiple processors in order to obtain a lower response time and a higher throughput from the system. Several alternative strategies exist for horizontally declustering a relation. However, the performance tradeoffs of these declustering strategies with respect to the different storage and access structures have not been previously examined. Thus, a database administrator for such a system has no choice but to guess how many processors a relation should be declustered across and what declustering strategy is most appropriate. These are important decisions since they have a significant impact on the response time and throughput of the system.

In this thesis, we evaluate the performance of the alternative declustering strategies for different selection query types in a multiuser environment and quantify the tradeoffs associated with each organization in the context of the Gamma database machine. Based on these results, we propose a new declustering strategy termed the **multidimensional** declustering strategy that is a better alternative than the existing declustering strategies. We present the design of this declustering strategy and compare its performance to the current declustering strategies provided by Gamma.

ACKNOWLEDGMENTS

One gets a few opportunities in a life time to publicly go on the record and acknowledge the contribution of other people to a piece of work as time consuming as a thesis. However, this is a mixed blessing. On one hand, I finally get the opportunity to thank the special people whose contributions have been invaluable to the quality of this thesis. On the other hand, I am almost certain that I have forgotten to acknowledge some of the people who have had a major impact on this thesis. I would like to apologize in advance to these people for being so forgetful.

I am most fortunate to have parents who have always been supportive of my decisions. The strength they have given me to make my own decisions has been an invaluable resource. They have installed in me the value of hard work to achieve excellence and to recognize and respect other people's good work. They have given first priority to my education and have never allowed circumstances beyond their control (I can recall many) to affect that priority. I can go on for pages about my parents, but no words can describe my love and respect for my father Arastoo Ghandeharizadeh and mother Mehrangiz Jamzadeh.

It has been my privilege to work and study with Professor David DeWitt. Technically, David has provided a very stimulating research environment with an incredible degree of freedom. He has allowed me to flourish at my own pace and to make my own share of mistakes. However, every time that I have stumbled, he has caught me in mid air and I am most grateful to him for that. He has significantly improved both my technical writing and presentation skills and has tried hard to refine them at every opportunity. He has provided a generous support and the opportunity to attend international conferences. Non-technically, David has been a good friend. He has tried to understand some of the most stressful situations of these graduate years and to make constructive suggestions. I would like to thank David for his enthusiasm, encouragement, and patience.

I would also like to thank both Miron Livny and Mike Carey for the many long discussions. I have learnt alot about performance modeling from both of them. They have been enthusiastic and supportive of my work and have encouraged me at every opportunity. Special thanks are due to Professor Meyer for his theoretical work on the design aspects of the Multidimensional Declustering Strategy. The opportunity to do joint work with Professor Meyer and his students has been a very exciting and educational experience. I am also grateful to Jeff Naughton for his constructive comments to improve the quality of this thesis.

The staff of Computer Sciences Department have provided significant support for the experimental vehicle of this thesis. Special thanks are due to Allan Bricker and Rick Rasmussen for maintaining and upgrading some of the crucial and tedious portions of the Gamma software through an endless and painful number of vendor upgrades and hardware changes. The experimental numbers presented in this thesis would have not been possible without their efforts. I would also like to thank Paul Beebe, Bruce Cole, Mitali Lebeck, Jim Luttinen, past and present members of the system lab for their exceptional quality of support and assistance.

I would like to thank some of my best friends for some of the memorable and craziest moments of my life. Kaishoon Basrai, Donovan Schneider, Ganesh Mani, Nance Kinne, and Kathy Hall have been an inspiration and have never failed to put a smile on my face (regardless of the number of revolvers to my head). I would especially like to thank Donovan for his friendship. I have had the luxury of his company during many sleepless nights under tight paper deadlines to our exciting travels to Japan and Australia.

I would also like to thank Scott Vandenberg, Samir Mahfoud, Eugene Shekita, Rajiv Jauhari, Hui Hsiao, Scott Leutenegger, Amarnath Mukherjee, Mike Franklin, Sanjay Krishnamurthi, Jonathan Yackel, Gary Schultz, and countless others at UW for their friendship.

Sheryl Pomraning has been an invaluable help to this research. I am especially thankful for her skills in the preparation of technical reports and complex gremlin figures.

I would also like to acknowledge my brothers Shahryar, Kourosh, and Kambiz for their cheer leading capabilities during these past years.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

In multiprocessor database machines, several forms of parallelism can be utilized to improve the performance of the system. First, parallelism can be applied by executing several queries simultaneously. This form of parallelism is termed **inter-query** parallelism. Second, **inter-operator** parallelism can be employed to execute several operators within the same query concurrently. For example, multiple processors could execute two or more relational join operators of a complex bushy join query in parallel. Finally, **intra-operator** parallelism can be applied to each operator within a query. For example, multiple processors can be employed to execute a single relational selection operator. In this thesis, we study the tradeoffs associated with exploiting intra-operator parallelism to execute the selection operator. The major reason for concentrating on the selection operator is that it is one of the basic operators of any query plan. Furthermore, it has a significant impact on the performance of a complex query since, if the appropriate degree of intra-operator parallelism cannot be provided for this operator, the performance and degree of intra-operator parallelism of the other operators in a complex query plan may severely be limited. This is especially true for database machines that utilize the concept of pipelining and data flow to execute complex queries - i.e., using inter-operator parallelism (e.g., Gamma [DEWI90], Bubba [BORA90], Volcano[GRAE89]).

In order to parallelize the selection operator, data must be horizontally declustered [RIES78, LIVN87] across multiple processors. The number of processors that a relation is declustered across determines the maximum degree of parallelism for this operator. However, parallelism is not for free; there are communications, startup, and termination overheads associated with parallelism. Furthermore, these forms of overheads increase as a function of the number of processors that a relation is declustered across and the algorithms employed to

schedule and terminate an operator. Thus, while increased declustering provides the potential for a higher degree of parallelism, additional declustering actually reduces throughput for certain classes of selection queries. The strategy used for declustering a relation strikes a compromise between these two conflicting factors by providing sufficient information to the optimizer to localize the execution of certain queries to a subset or even a single processor, minimizing the overhead incurred in its execution.

The Gamma database machine currently provides the database administrator (DBA) with three alternative partitioning strategies: 1) round-robin, 2) hash, and 3) range partitioned with administrator-specified key values. In the first strategy, the tuples in a relation are distributed in a round-robin fashion among the processors. In effect, this results in a completely random placement of tuples with a uniform distribution of tuples across the processors. In addition, this declustering strategy causes the degree of parallelism for a selection query to always equal the number of processors that a relation is declustered across. In Gamma this strategy is only used for storing the result tuples of a query back to the database. In the hash partitioning strategy, a randomizing function is applied to the partitioning attribute of each tuple to select a home processor for that tuple. With this declustering strategy, a single processor is employed to execute queries with an equality predicate on the partitioning attribute. However, it directs queries with a range predicate to all the processors that contain the fragments of the referenced relation. In the last strategy, the DBA specifies a range of key values for each partition or processor. This strategy gives a greater degree of control over the distribution of tuples across the processors. Furthermore, the execution of queries with either an equality or a range predicate on the partitioning attribute can always be localized to those processor(s) that contain the relevant tuples.

Several commercial systems and research prototypes also support one or more of these declustering strategies. For example, Bubba supports both range and hash declustering strategies [BORA90], while the Non-Stop SQL product from Tandem [TAND88] supports only the range declustering strategy, and the DBC/1012 database machine from Teradata [TERA85a, TERA85b] can only hash decluster a relation.

For each relation, the DBA must also select a storage structure. Gamma currently provides three possible storage/access structure alternatives : 1) heap, 2) clustered index, and 3) non-clustered index. In the first strategy, the tuples are stored in the order that they arrive at a processor. For the clustered index access structure, tuples are first sorted by their key value and an index is constructed on the key attribute. For the non-clustered index structure, the index order is different than the order of the key value of the relation (the file may be organized as a heap, hashed on the same or a different attribute, or sorted on a different attribute).

In this thesis we study the impact of alternative partitioning strategies and storage organizations on different selection query types in a multiuser environment and quantify the tradeoffs of each organization in the context of the Intel iPSC/2 hypercube version of the Gamma database machine. Based on these results, we propose a new declustering strategy (termed the **multidimensional** declustering strategy) that provides superior support for a wider range of selection queries and alternative distributions of the partitioning attribute values than provided by the current Gamma partitioning strategies.

1.1. Motivation

In this section, we motivate this study of the alternative partitioning strategies and storage/access structures. We will also briefly describe the new partitioning strategy and discuss its advantages. The following stock market database is used as an example throughout our discussion. This database consists of the following relations:

BROKER (Broker_ID, name, Brokerage-firm)

STOCK (ticker-symbol, name, price, highest, lowest, closing, opening, P/E)

BID (BID_#, ticker-symbol, Broker_ID, price, quantity, time)

ASK (ASK_#, ticker_symbol, Broker_ID, price, quantity, time)

TRADE (TRADE_#, ticker_symbol, Buyer_Broker_ID, Seller_Broker_ID, price, quantity, date, time)

The BROKER relation identifies persons allowed to trade in the stock market and consists of

the following attributes: Broker_ID, which uniquely identifies a broker; the name of the broker; and the brokerage firm he is associated with. The STOCK relation consists of a ticker_symbol which uniquely identifies a company (e.g., AXP represents American Express), the name of the company (e.g., American Express), the price, highest, lowest, closing and opening price of a share of that company, and the price earning estimate attribute (P/E) which provides the potential buyers with an approximate appraisal of a company[1]. The BID relation identifies the brokers who have entered bids to buy a certain number of shares of a company for a certain price. Conversely, the ASK relation identifies the brokers who are selling a quantity of shares of a given company for a certain price. The TRADE relation maintains the matching bids and asks that have resulted in a trade.

To motivate the importance of supporting a variety of alternative partitioning strategies and storage/access structures, consider the STOCK relation. Assume that 80% of the queries (termed type one queries) retrieve a single record by specifying the ticker_symbol of that company (e.g., retrieve STOCK.all where STOCK.ticker_symbol = AXP). The other 20% of the queries (termed type two queries) retrieve the companies whose price-to-earning ratio is within a specified range of values (e.g., retrieve STOCK.all where STOCK.P/E > 10). For this relation and workload, the optimal horizontal partitioning strategy is hash partitioning on the ticker_symbol attribute and the appropriate access structure at each processor is a non-clustered index structure on the ticker_symbol and a clustered index structure on the P/E attribute. By hash partitioning on the ticker_symbol attribute, the query optimizer can direct the type one queries to the processor containing the desired STOCK record corresponding to the specified company. After the query arrives at the processor, the processor uses the non-clustered index on the ticker_symbol attribute to efficiently retrieve the specified tuple. The type two queries will be directed to all the processors, where they will be processed using the

---

[1]To be completely accurate, P/E specifies how much profit the company has earned per share in the past quarter (this value will be negative if the company has experienced a loss).

clustered index on the price attribute (avoiding a sequential scan of the entire relation).

The type one queries retrieve a single tuple and thus only one processor contains the qualifying tuple. By directing this query to a single processor, the communication overhead associated with scheduling and terminating the query is minimized, resulting in the best system response time. In addition, when directed to a single processor, only minimal resources are consumed to execute this query. If the query was directed to additional processors, these processors would have consumed CPU cycles and I/O bandwidth only to determine that their fragment of the relation does not contain the qualifying tuple. When this query is directed to the processor with the relevant tuple, the other processors are freed to execute some other concurrently executing query, increasing the throughput of the system.

However, if a query retrieves a large number of tuples and its execution time is high enough to render the overhead of using parallelism insignificant, utilizing a high degree of parallelism will be most beneficial. This is because additional processors will improve the response time of such queries without adversely affecting the throughput of the system. In addition, the execution of such queries on a single processor might result in the formation of hot spots and bottleneck processors in a multiuser environment[2].

In this example, the partitioning requirements of the two query types in the workload did not conflict with one another. However, sometimes the preferred partitioning requirements of the queries in the workload do conflict and one of the alternatives must be selected. This is not a simple decision since the objective is to provide the best response time and throughput for the whole workload, not just one of the query types in the workload. To illustrate this, consider the TRADE relation. Assume that 60% of the queries (termed type one queries) retrieve a record by specifying the ticker_symbol of the company (e.g., retrieve TRADE.all where TRADE.ticker_symbol = AXP). The other 40% of the queries (termed type two queries) retrieve companies which have been traded in a specified price range (e.g., retrieve TRADE.all where

---

[2] We will demonstrate these effects in Chapter 5.

TRADE.price > 10 and TRADE.price < 12). Furthermore, assume that the selectivity factor of the type two queries is very low. The preferred partitioning strategy for the type one query is hash partitioning on the ticker_symbol attribute, while the preferred partitioning strategy for the type two query is range partitioning on the price attribute. The preferred storage structure at each processor is a non-clustered index on the ticker_symbol attribute and a clustered index on the price attribute.

There are several factors that determine whether the TRADE relation should be hash partitioned on the ticker_symbol attribute or range partitioned on the price attribute. Two of these factors are presented for illustration purposes in this section. The first factor is the frequency of occurrence of each query type. For example, the type one query has a 20% higher frequency of occurrence than the type two query. Therefore, it might be more advantageous to obtain the best response time and throughput for the type one query by hash partitioning the TRADE relation on the ticker_symbol attribute. The second factor is the distribution of the partitioning attribute values. For instance, the stock market periodically observes a takeover bid for a company by a set of other companies or investors. Under these circumstances, the stocks of the companies involved will be traded heavily. Consequently, a large number of the records in the TRADE relation will have the same value in the ticker_symbol attribute. By hash partitioning on this attribute, a large number (possibly the majority) of the tuples of this relation will be directed to a single processor, resulting in a higher processing workload on this processor than the other processors. Furthermore, under heavy system loads, this processor will almost certainly become a bottleneck for the system since it must perform more processing than the other processors. In this thesis we will quantify the tradeoffs involved in choosing among the alternative partitioning strategies under different workload characteristics.

As a result of our performance evaluation and study of the alternative partitioning strategies, we will introduce a new partitioning strategy termed the **multidimensional partitioning strategy**. This partitioning strategy horizontally declusters a relation based on several attributes while maintaining a uniform distribution of the workload across the processors. It is introduced to solve the limitations of the existing partitioning strategies. To illustrate some of the

limitations of the existing partitioning strategies and how the multidimensional partitioning strategy resolves them, consider the BID relation. In general a broker will bid on the shares of a company depending on the quantity of shares he is willing to buy. He will offer a lower price per share if he is purchasing a large quantity of shares than if he is buying only a few shares. Therefore, there might be several bids for a company by a single broker with different prices and quantities. Assume that 40% of the queries (type one queries) retrieve the outstanding bids on a company for a certain price (e.g., retrieve BID.all where BID.ticker_symbol = AXP and BID.price = 15). The other 60% of the queries are submitted by speculating brokers who own a large quantity of different stocks. These brokers access the BID relation to retrieve information on the other brokers who are buying either a large quantity of shares or are offering a large price per share of any company (e.g., retrieve BID.all where BID.quantity > 10,000 or BID.price > 110). The preferred partitioning strategy for the type one query is hash partitioning on the ticker_symbol attribute. There is no real preferred partitioning strategy for the type two query, since both range and hash partitioning strategies will direct any selection query with a predicate composed of multiple disjuncts to all processors for execution (similar to round-robin). This is because range and hash partitioning strategies can horizontally partition a relation based on only one attribute. This is a major limitation since even if the selectivity factor of type two queries is very low, it will still be executed by all the processors storing tuples from the BID relation.

With the multidimensional partitioning strategy, the BID relation can be partitioned on the ticker_symbol, price, and quantity attributes. This provides the capability for localizing the execution of any query accessing either of these three attributes or a combination of them with a disjunctive or a conjunctive predicate. To illustrate this, consider the two query types presented above. For the first query type, the optimizer determines the set of processors needed for the execution of each conjunct using the information provided by the multidimensional partitioning strategy. The processors resulting from the intersection of these two sets will participate in the execution of this query type. Furthermore, if the range of the values of the price attribute is known to be greater than 20, then the set of processors participating in

the execution of the predicate BID.price=15 will be null. Therefore, the intersection of the set of processors participating in the execution of this query will be null. With this information, the query optimizer does not need to submit this query for execution at all, since it already knows that no tuples will satisfy the predicate of the query. With the second query type, the set of processors needed for the execution of each disjunct is determined. Those processors in the union of these two sets will participate in the execution of this query type. Thus, the multidimensional partitioning strategy can partition a relation based on several attributes, and it can localize the execution of a greater variety of queries than either the hash or range partitioning strategies.

The objective of the multidimensional declustering strategy is not to employ a single processor to execute all selection queries. Rather, it is to provide the appropriate degree of parallelism for a set of queries accessing a relation by utilizing the characteristics of these queries in order to decluster the relation. By declustering a relation in this manner, the multidimensional declustering strategy provides effective support for small relations and for relations whose partitioning attribute values are not uniformly distributed.

1.2. Goals

The primary objectives of this dissertation are to quantify the performance tradeoffs associated with the alternative declustering strategies, and to report on the design, implementation, and performance evaluation of the multidimensional partitioning strategy. This partitioning strategy declusters a relation by analyzing the characteristics of the set of queries accessing the relation. We will evaluate the performance of the one dimensional version of the multidimensional declustering strategy, comparing it to the existing declustering strategies. When the number of dimensions is equal or greater than two, the assignment of fragments to the processors becomes a critical issue as the performance of the system will degrade significantly if the fragments are not assigned to the appropriate number of processors. We have designed a heuristic for assigning the fragments to the processors. In this thesis the performance of this heuristic and how closely it approximates the optimal assignment of fragments to processors is

also presented.

1.3. An Overview of the Thesis

In Chapter 2, we describe previous work that is related to this thesis. Chapter 3 provides an overview of the Gamma multiprocessor database machine and its execution paradigm for the relational algebra operators. In Chapter 4, the workload model used for evaluating the performance of the alternative partitioning strategies is described. Chapter 5 presents the performance of the alternative partitioning strategies for the selection workload. In Chapter 6, we present the design of the multidimensional partitioning strategy (MDPS). The performance of the one dimensional version of MDPS is compared to the other declustering strategies in Chapter 7. Our conclusions and future research directions are contained in Chapter 8.

CHAPTER 2

A SURVEY OF RELATED WORK

There have been a number of earlier studies of the problem of file distribution in multiprocessor database machines. Some have proposed new declustering strategies [DU82, KIM88, COPE88, PRAM89], while others have analyzed the impact of alternative declustering strategies [LIVN87, GHAN90a]. Studies that are concerned with the multi-disk architecture [DU82, LIVN87] are different and not directly relevant to this study. In order to make this distinction clear, we use the two dimensional file system of the XPRS shared-memory multiprocessor database management system [STON88] as an example. XPRS uses a disk array for mass storage (based on the RAID design [PATT88]). It supports intra-query parallelism by fragmenting a single relation into multiple files. Like range declustering in Bubba and Gamma, each relation is fragmented using a distribution criteria [STON88], e.g.:

```
STOCK where price < 20                          to file 1
STOCK where price >= 20 and price <= 40         to file 2
STOCK where price > 40                          to file 3
```

Furthermore, each file is organized as a number of extents. In traditional file systems, an extent corresponds to a sequential number of blocks on a disk drive. In XPRS, each extent is two dimensional: one dimension corresponds to the number of disks that contain an extent (W_i) while the second dimension corresponds to the number of tracks on each disk (S_i). The choice of W_i and S_i is suggested by the application software [STON88].

In an environment similar to that of XPRS, the alternative declustering strategies would define how each file should be declustered and the number of files a relation must be fragmented into and not the values of S_i or W_i. Determining the values of W_i and S_i is a separate problem with a different set of constraints that is beyond the scope of this thesis.

[COPE88] examined the problem of data placement in Bubba [ALEX88, BORA88, BORA90]. This study introduced concepts such as *heat* and *temperature* which are crucial to understanding the issues relevant to this problem. However, the major thrust of this study was to maximize the throughput of the system. In order to achieve this objective, this study focused on two issues: 1) balancing the load across the processors in the environment, and 2) whether to place the data on disk or cache it permanently in memory. Due to the complexity of the problem (NP-complete [GARE71]), a heuristic approach was used to solve the problem. This study did not consider the impact of alternative declustering strategies on the throughput of the system. In addition, the algorithms presented in [COPE88] do not attempt to minimize the response time of a transaction.

[KIM88, PRAM89] presented an optimal file declustering strategy for partial match retrieval queries. The major contribution of this study was the development of a new approach termed the Fieldwise eXclusive-or (FX) distribution method which maximizes data access concurrency in the presence of partial match queries. A limitation of FX distribution is that it does not provide support for range queries. Furthermore, the distribution algorithm is dependent on the distribution of the attribute values used to decluster a relation (similar to the range and hash declustering strategies).

This thesis differs from the earlier efforts in that: 1) it evaluates the performance of the alternative declustering strategies in the context of an actual shared-nothing multiprocessor database machine and 2) it describes a new declustering strategy that partitions a relation by analyzing the characteristics of the queries accessing that relation and the processing capability of the system. By taking these factors into consideration, this new declustering strategy attempts to provide the appropriate degree of intra-query parallelism for the queries in the workload. A major difference between the multidimensional partitioning strategy (MDPS) and the hash, range, and FX declustering strategies is that the MDPS declusters the relation independently of the distribution of the partitioning attribute values. In addition, the MDPS can decluster the relation based on several attributes.

CHAPTER 3

OVERVIEW OF THE GAMMA DATABASE MACHINE

This chapter presents a brief overview of the Gamma database machine. Included in this discussion is the current hardware configuration and software techniques used in implementing Gamma. For a complete description of the Gamma database machine see [DEWI90].

Briefly, Gamma is a software database machine based on a shared-nothing architecture in which processors do not share disk drives or random access memory and can only communicate with one another by sending messages through an interconnection network.

3.1. Hardware Configuration

Presently, Gamma runs on a 32 processor iPSC/2 Intel hypercube [INTE88]. Each processor is configured with an Intel 80386 processor, 8 megabytes of memory, and a 333 megabyte MAXTOR 4380 (5 1/4") disk drive. Each disk drive has an embedded SCSI controller which provides a 45K byte RAM buffer that acts as a disk cache on read operations. In a single user environment, the SCSI cache speeds up the execution of sequential scan queries by almost a factor of two. However, in a multiuser environment, its effect is minimal because the probability of two successive disk accesses being sequential is quite low. Furthermore, a disk request that results in a seek completely invalidates the contents of the SCSI cache. We will repeatedly see the impact of the SCSI cache on the performance of the different selection queries in Chapter 5 of this thesis.

3.2. Software Overview

Physical Database Design

In Gamma, relations are **horizontally declustered** [RIES78, LIVN87] across all disk drives in the system. The key idea behind horizontally declustering each relation is to enable the database software to exploit all the I/O bandwidth provided by the hardware. The query language of Gamma provides the user with three alternative declustering strategies: round-robin, hash, and range partitioned with user-specified placement by key value. Once a relation has been declustered, Gamma provides the normal collection of relational access methods including both clustered and non-clustered indexes. When the user requests that an index be created on a relation, the system automatically creates an index on each fragment of the relation. Unlike VSAM [WAGN73] and the Tandem file system [TAND85], Gamma does not require the clustered index for a relation to be constructed on the partitioning attribute.

Query Execution

Gamma uses traditional relational techniques for query parsing, optimization [SELI79, JARK84], and code generation. Queries are compiled into a tree of operators with predicates compiled into machine language. After being parsed, optimized, and compiled, the query is sent by the host software to an idle scheduler process through a dispatcher process. The scheduler process, in turn, activates operator processes at each query processor selected to execute the operator. The task of assigning operators to processors is performed in part by the optimizer and in part by the scheduler assigned to control the execution of the query. For example, the operators at the leaves of a query tree reference only permanent relations. Using the query and schema information, the optimizer is able to determine the best way of assigning these operators to processors. The results of a query are either returned to the user through an ad-hoc query interface or through an embedded query interface to the program from which the query was initiated.

In Gamma, the algorithms for all operators are written as if they were to be run on a single processor. As shown in Figure 3.1, the input to an Operator Process is a stream of tuples and the output is a stream of tuples that is demultiplexed through a structure we term a **split**

table. After being initiated, a query process waits for a control message to arrive on a global, well-known control port. Upon receiving an operator control packet, the process replies with a message that identifies itself to the scheduler. Once the process begins execution, it continuously reads tuples from its input stream, operates on each tuple, and uses a split table to route the resulting tuple to the process indicated in the split table. Consider, for example, the case of a selection operation that is producing tuples for use in a subsequent join operation. If the join is being executed by N processes, the split table of the selection process will contain N entries. For each tuple satisfying the selection predicate, the selection process will apply a hash function to the join attribute to produce a value between 1 and N. This value is then used as an index into the split table to obtain the address (e.g. machine_id, port #) of the join process that should receive the tuple. When the process detects the end of its input stream, it first closes the output streams and then sends a control message to its scheduler indicating that it has completed execution. Closing the output streams has the side effect of sending *end of stream* messages to each of the destination processes. With the exception of these three control messages, execution of an operator is completely self-scheduling. Data flows among the processes executing a query tree in a dataflow fashion. If the result of a query is a new

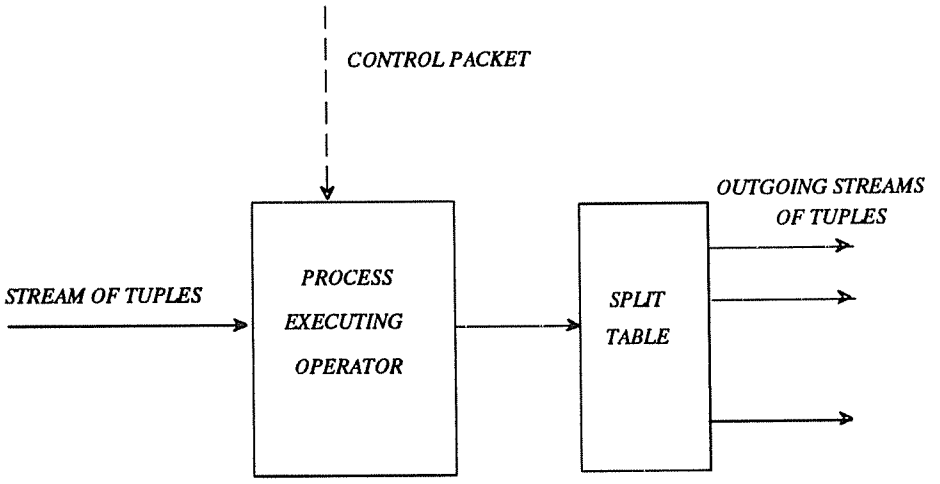


Figure 3.1

relation, the operators at the root of the query tree distribute the result tuples on a round-robin basis to store operators at each processor. The store operators assume the responsibility for writing the result tuples to disk.

Operating and Storage System

Gamma is built on top of an operating system called **NOSE**, that was developed specifically for supporting database management systems. NOSE provides lightweight processes with shared memory.

File services in Gamma are based on the Wisconsin Storage System (WiSS) [CHOU85a]. These services include structured sequential files, B^+ indices, byte-stream files as in UNIX, long data items, a sort utility, and a scan mechanism. A sequential file is a sequence of records. Records may vary in length (up to one page), and may be inserted and deleted at arbitrary locations within a sequential file. Optionally, each sequential file may have one or more associated indices. The index maps key values to the records of the sequential file that contain a matching value. Furthermore, one indexed attribute may be used as a clustering attribute for the file. The scan mechanism is similar to that provided by System R's RSS [ASTR76] except that predicates are compiled into machine language. WiSS ensures the serializability of the concurrently executing queries by using a two-phase locking [GRAY78] mechanism.

CHAPTER 4

WORKLOAD DEFINITION

This chapter describes the workload model used for evaluating the performance of the alternative partitioning strategies. We will continue using the stock market database for illustration purposes throughout this chapter.

In order to evaluate the performance of the alternative declustering strategies, both the response time of the queries and the processing effort of the system (total amount of CPU, I/O bandwidth and other resources in the environment) required to execute these queries must be taken into consideration since these two factors do not necessarily coincide in a multiprocessor database machine. To illustrate this, recall the STOCK relation. Assume that the query accessing this relation retrieves 10 tuples by specifying a range of values for the price attribute (e.g., retrieve STOCK.all where STOCK.price > 10 and STOCK.price <= 20). Furthermore, assume that no indices exist on the price attribute resulting in a sequential scan of the STOCK relation at each processor. Consider the performance of hash partitioning compared with that of range partitioning for this query. Since the hash partitioning strategy cannot localize the execution of range queries, the query will be directed to all the processors, resulting in a sequential search at each processor. If the relation is instead range partitioned on the price attribute, then the query will be directed to only one of the processors. The response time of the query with each of these alternative partitioning strategies would be the same (assuming the sequential search of the relation is expensive enough to render the overhead of scheduling and committing the query at multiple processors insignificant). However, the amount of resources (CPU and I/O) consumed by the hash partitioning strategy would be P times the amount of resources used by the range partitioning strategy (where P is the total number of processors used to execute the query) .

The best way of capturing the effect of the resources consumed by a particular partitioning strategy is by analyzing the impact of a number of concurrently executing queries on the response time of the individual queries and the throughput of the system as a whole. To illustrate this, assume two queries with a range predicate on the price attribute are submitted to a system with *P* processors. With the hash partitioning strategy, the two queries will compete for CPU cycles and I/O bandwidth at each processor, causing the response time[1] of each query to increase. With the range partitioning strategy, the two queries are likely to be directed to two different processors, resulting in no contention for CPU or I/O bandwidth at either processor. Thus, the response time of each query will appear as if it executed by itself. Furthermore, the throughput of the system will be doubled since two queries were executed in the elapsed time required for one. Therefore, to accurately evaluate the performance of alternative partitioning strategies, the throughput and response time of the system for each individual query must be evaluated while the number of concurrently executing queries is varied (we term this variable the **multiprogramming level**).

But what happens to the throughput of the system with the hash partitioning strategy, or when the range partitioning strategy fails to direct the two queries to different processors? Under these circumstances, the major determining factor is the degree of data sharing among the two concurrently executing queries. To illustrate this, consider both the STOCK and BID relations. Assume that the query accessing the BID relation retrieves records using a range predicate on the price attribute (retrieve BID.all where BID.price > 10 and BID.price <= 20). The workload of the STOCK relation is the same as described above. Furthermore, assume that no indices exist on any attribute of the BID and STOCK relations. Two queries are submitted to the system, and one of the queries accesses the STOCK relation while the other one accesses the BID relation. With the hash partitioning strategy, both queries will be directed to

---

[1]The next paragraph will discuss the throughput of the system with this partitioning strategy.

all the processors. Not only will these two queries compete for CPU cycles and I/O bandwidth at each of the processors, but they will also compete for buffer pool pages. In addition, the two queries will result in random disk seeks to different areas of the disk drive instead of the series of sequential disk seeks that would have occurred had each been submitted individually. This contention for resources will increase the response time of the individual queries and will almost certainly result in a lower throughput from the system.

On the other hand, assume that the two queries access the same relation (say the STOCK relation). In this case, there is a possibility of cooperation between the two queries. Each query does some processing and then does an I/O (disk request). Assume that these two queries are started at approximately the same time. This will result in query 1 bringing a data page in and processing it. While query 1 goes to the disk to bring the next data page into memory, query 2 could process the page which was just processed by query 1. Query 2 might compete with query 1 for CPU cycles depending on the CPU processing time per page and the time taken by the disk to transfer a page into memory. If the CPU processing time is less than the disk transfer time per page, then query 2 will not compete for either of the resources directly. However, if the CPU processing time per page is greater than the disk transfer time, then the two queries will compete for CPU cycles. In this scenario, the throughput of the system actually increases since the queries are cooperating to a certain degree rather than competing. Thus, the degree to which the throughput of the system is affected with a partitioning strategy is dependent on the degree of data sharing between the concurrently executing queries (we term this variable the **degree of data sharing**).

Thus far, we have motivated the necessity for a multiuser workload to evaluate the performance tradeoff associated with the alternative partitioning strategies. We also discussed the need to study the effect of changing the multiprogramming level and the degree of data sharing. But three important questions remain to be addressed: 1) what mix of queries should be used, 2) how are these different variables controlled by a workload, and 3) are there any existing workload models that can satisfy all of our workload requirements? Currently, two multiuser workload definitions exist: 1) the Wisconsin multiuser benchmark, used for evaluating

the performance of relational database management systems and database machines [BORA84], and 2) the transaction processing benchmark (TP1) [ANON84]. In Sections 4.1 and 4.2 we will describe each of these workloads/benchmarks and discuss why each is insufficient for evaluating the performance of the alternative partitioning strategies. In Section 4.3 a general description of a new multiuser workload model and its parameters is presented.

4.1. Wisconsin Multiuser Benchmark

The Wisconsin multiuser benchmark was the first attempt at designing a standard workload to evaluate the multiuser performance of relational database management systems. Its evaluation criteria was the response time and throughput for four different query types. This benchmark identified three principal factors that affect performance in a multiuser environment: 1) multiprogramming level, 2) degree of data sharing, and 3) query mix selection. The multiprogramming level refers to the number of queries being executed concurrently. The degree of data sharing defines the number of concurrently executing queries accessing the same relation. It was introduced to evaluate the effectiveness of buffer pool management algorithms of different database systems. A scale of 0% to 100% was used as a measure of the degree of data sharing. When the degree of data sharing was 0%, all concurrently executing queries would reference disjoint subsets of the database. When the degree of data sharing was 100%, all concurrently executing queries referenced the same subset of the database. For less than a 100% degree of data sharing, the number of relations accessed was a function of the multiprogramming level. Query mix selection (the third factor) dealt with devising a small set of representative queries which had the capability of evaluating the performance of different database systems. Resource utilization was used as the basis for forming this small representative set. Basically, the observation was made that database queries consume two main resources: CPU cycles and I/O bandwidth. Consumption of each of these resources was classified into "low" and "high". Based on this classification, four types of queries were introduced: low CPU and low I/O, low CPU and high I/O, high CPU and low I/O, and high CPU and high I/O. The performance of the system was then analyzed for each query type and several

different mixes of query types under different degrees of data sharing and multiprogramming levels.

The model deserves recognition for introducing a methodology for evaluating the performance of different database systems. At first glance, this model appears ideal for the performance evaluation of the alternative partitioning strategies since it has the potential of satisfying all of our workload requirements. However, after a more thorough analysis of this model, we discovered the following flaw in its definition. For degrees of data sharing less than 100%, the physical size of the database changed as a function of the multiprogramming level. This meant that as the multiprogramming level increased, the physical size of the database became larger. Unfortunately, larger database sizes result in longer average seek times for the disk drive. This concerned us, since if an effect is observed when changing the multiprogramming level from x to x+y for less than 100% data sharing, this effect cannot be contributed solely to the change in the multiprogramming level. Rather, the effect must be attributed to both the change in the multiprogramming level and the corresponding change in the database size. Although this problem might not appear that significant, in Section 4.3 we will present the design of a modified version of this benchmark that achieves all of our workload requirements while solving this problem.

4.2. Transaction Processing Benchmark

The transaction processing benchmark (TP1) [ANON84] was an attempt by several members of the database community to develop a standard metric for evaluating the performance of a transaction processing system. The benchmark suggested three performance measurement criteria: 1) response time: the elapsed time of a transaction, 2) cost: a five year capital cost of vendor supplied hardware and software used to support the database, and 3) throughput: the number of transactions processed per second with 95% of the transactions having less than one second response time.

Three basic benchmarks were designed for the performance evaluation of a transaction processing system. The first benchmark was "scan", a batch execution of a transaction which

did a sequential scan and update of one million records (actually 1000 batches of 1000 records each). The second benchmark was "sort", where a million tuple relation was sorted. The last benchmark was "DebitCredit", in which a set of transactions accessed the database to debit a bank account, do the standard double entry book keeping and then reply to the terminal [ANON84]. The DebitCredit benchmark was run under the constraint that 95% of the transactions must execute in less than one second. The first two benchmarks measured the input/output performance of the system. This evaluation was made based on the response time and the cost of executing each of these transactions on the system. The DebitCredit was a very simple transaction processing application designed to measure the throughput and cost of the system.

The objective and goals of the TP1 benchmark set were well defined. Simplicity and portability were two of its major objectives. It achieved both of these objectives and introduced a standard benchmark (DebitCredit) for evaluating different database systems designed for transaction processing (banking applications). However, while TP1 is partially sufficient for evaluating the performance of the alternative partitioning strategies, it is not complete. Each alternative partitioning strategy introduces a set of capabilities to a multiprocessor database machine. The DebitCredit transaction would evaluate only a single aspect of these alternative capabilities. To illustrate this, consider the hash and range partitioning strategies. The hash partitioning strategy directs all single tuple retrieval queries to a single processor, while the range queries are executed on all the processors (no matter how small or large the selectivity of the range query). However, the range partitioning strategy localizes the execution of both the single tuple and range selection queries. These are obviously two different capabilities introduced by these two alternative partitioning strategies. Since the DebitCredit transaction would only consider the performance of the single tuple retrieval and update queries, both the range and hash partitioning strategies would exhibit similar levels of performance for this query. Therefore, the DebitCredit transaction would fail to distinguish that these two partitioning strategies provide different execution paradigms for range queries. Thus, TP1 is not complete for a thorough evaluation of the alternative partitioning strategies.

4.3. General Methodology Behind The New Multiuser Workload Definition

The workload model that we have designed for evaluating the performance of the alternative partitioning strategies is very general. It was not made specific because doing so would have limited the applicability of the benchmark while making the experiments appear very dependent on the specific characteristics of the benchmark. Rather, the goal was to design a general workload model with the potential for scalability.

In the introduction to this chapter, three principal factors were identified that affect the performance of the queries with the alternative partitioning strategies: 1) multiprogramming level, 2) degree of data sharing, and 3) query mix. Each factor defines an axis in the performance space that can be varied while keeping all the other factors constant. Below, we present each of these factors and discuss a general methodology developed for analyzing them. But first, we establish the criteria used for evaluating the performance of the alternative partitioning strategies.

4.3.1. Performance Measurements

In various sections in this thesis, the response time and throughput of the system for various Gamma configurations is presented. Specifically, the response time and throughput of the system are used as the performance metrics for evaluating the alternative partitioning strategies. Due to the random nature of the workload model, each experiment has to run long enough to provide meaningful measurements. In addition, statistical analysis of measurements is required to establish the validity of the results. There are three alternatives for estimating confidence intervals: batch-means, independent replications, and the regenerative method [SARG76]. We choose batch-means analysis for establishing confidence interval. Analysis of the response time measurements indicates that all the confidence intervals are within ±2.5% of the mean response time (95% confidence interval). The throughput of the system was computed by dividing the multiprogramming level with the average response time of the system (i.e., $\frac{multiprogramming\ level}{average\ response\ time}$).

4.3.2. Multiprogramming Level

The multiprogramming level refers to the number of concurrently executing queries. The term "executing" should be made specific. There are four stages to query execution : 1) parsing, 2) optimizing and access planning, 3) executing the query, and 4) receiving the results of the query. The parsing and access path selection stages of query execution were removed by embedding the query in a program and compiling it. Therefore, the term "executing" here refers to the last two stages of query execution.

4.3.3. Degree of Data Sharing

This variable provides the means of controlling the degree of data sharing at the buffer pool area and the degree of data contention at the concurrency control manager of the system. If there is a high degree of data sharing between the concurrently executing queries, then the CPU will almost certainly be the only resource these queries would directly compete for. However, a high degree of data sharing can translate to a high degree of data contention in the presence of update queries. This is because the update queries attempt to obtain exclusive access to portions of the database that are needed by the other concurrently executing queries. This will result in blocking, and in the case of deadlocks, aborting one or more queries. On the other hand, a low degree of data sharing results in competition for CPU, buffer pool pages, and I/O bandwidth among the concurrently executing queries, but the degree of lock contention is significantly reduced.

How should the different degrees of data sharing be supported? One might argue that it should be defined as a function of the percentage of buffer pool hits observed for an interval of time. This is not reasonable since the variable that we termed the degree of data sharing was introduced to evaluate the performance of the different partitioning strategies. It was not meant to be a dependent variable defined based on the characteristics of the underlying workload (like the response time or the throughput).

We implemented the different degrees of data sharing by controlling the frequency of accesses made to the relations in a fixed size database. The decision was made to support two

degrees of data sharing: low, and high[2]; each of which was meant to be an approximation of the 0%, and 100% degrees of data sharing respectively, found in the Wisconsin multiuser benchmark. In order to implement these two degrees of data sharing, a fixed database size with **m** relations was designed. The cardinality of each relation was kept at **n** tuples (total size of database in bytes = m * n * size of each tuple). Below, we will describe how the frequency of accesses made to the relations was manipulated in order to provide the desired effect.

We used the method presented in [LIVN87] to support the different degrees of data sharing. The method is based on a normal distribution of access to the relations in the database. Assuming that the relations are numbered from 1 to m, the probability of access to the relation R_i is:

$$P_i = \frac{P\left[\frac{i-1}{m}, \frac{i}{m}\right]}{P(0, 1)}$$

where $P(a, b) = P[a \leq x < b$; x is a variable based on the normal distribution with mean zero and standard deviation $\sigma]$. σ determines the degree of non-uniformity to the relations in the database. In reality, σ can be viewed as a parameter that controls the distribution of access to the relations. Below, we explain how the different distributions of accesses are used to support the two different degrees of data sharing.

We interpreted a low degree of data sharing to mean a database in which the frequency of access to each relation is the same. Thus, for a single user accessing a database of m relations, each relation in the database is accessed with a probability of 1/m. We approximated a uniform distribution of access by choosing a value of 3 for σ. Figure 4.1 presents the frequency of accesses made to the relations for 3 different database sizes. The x-axis refers to the

---

[2] We did not consider a medium degree of data sharing because the results obtained in [GHAN90a] and [GHAN90b] revealed that for variety of queries, the performance of the system for medium degree of data sharing always falls somewhere in between low and high degrees of data sharing. Furthermore, no interesting results or conclusions could be drawn from the performance of the system for a medium degree of data sharing.

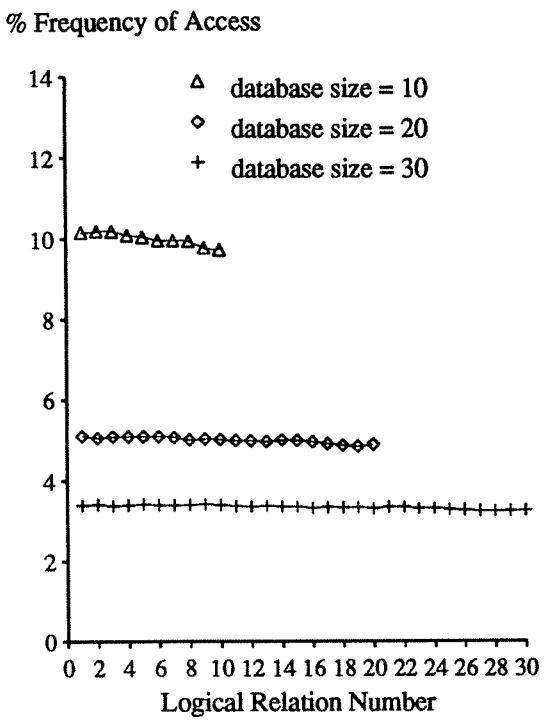


Figure 4.1

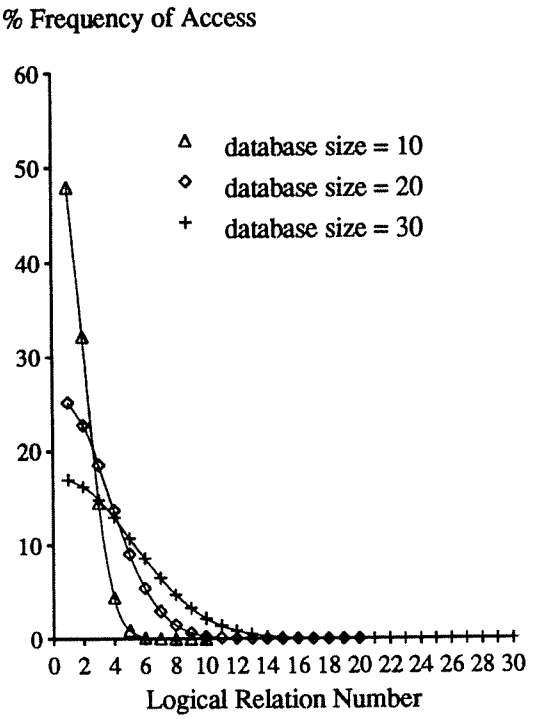


Figure 4.2

relation being accessed (relations in the database are logically numbered from 1 to m). The y-axis corresponds to the frequency of accesses to a relation. The different curves correspond to database sizes of ten, twenty, and thirty relations. As the number of relations (m) is increased, the frequency of access to each relation decreases proportionally. The frequency of access to each relation is not exactly $\frac{100}{m}$ because a random number generator is used to create the normal distribution.

A high degree of data sharing is interpreted to mean a database in which a small number of the relations are accessed most frequently, e.g., if 80 percent of accesses are made to 20 percent of the database. In order to support such a distribution of accesses, we used a value of 0.156 for σ. The distribution curves for three different database sizes are presented in Figure 4.2. The y-axis and x-axis have the same meaning as above. An observation to be made from these figures is that as the number of relations in the database is increased, the distribution becomes less and less skewed toward logical relation number one. The reason is that as

the database size is increased, the 20% region of the database is proportionally increased. Consequently, 80% of the accesses must now be shared among a larger number of relations, resulting in a lower frequency of access to logical relation number one.

Figure 4.3 presents the two different distributions of accesses to a database consisting of ten relations (m=10). This was done mainly for illustration purposes since the scale of the y-axis was not kept constant across Figures 4.1 and 4.2. This figure provides a comparison point for the two different degrees of data sharing.

In the proceeding paragraphs, we presented our definition of low and high degrees of data sharing and the techniques used to achieve these definitions. The frequency of accesses chosen for each region of the database within a given degree of data sharing is very arbitrary (e.g., 20% of the database is accessed 80% of the time for the high degree of data sharing). The point is that we chose these patterns of access to support what we thought best represented high and low degrees of data sharing.

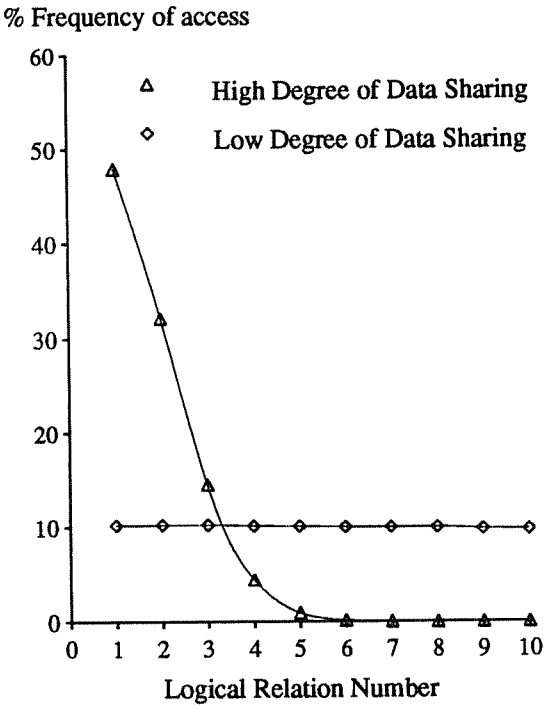


Figure 4.3

The value of m, the number of relations in the database, remains to be addressed. What value of m should be used? The answer is simple: m should be as large as necessary to approximate a 0% degree of data sharing. If it is desirable to simulate exactly a 0% degree of data sharing, then the number of relations in the database (m) should be set to the maximum multiprogramming level used for the experiments. Furthermore, the function which creates the pattern of access to the database must ensure that two concurrently executing queries do not access the same relation (by definition of the strict 0% data sharing from [BORA84]). How is this different than the definition of 0% data sharing in the Wisconsin multiuser benchmark? An example will make the distinction clear. Assume that the maximum multiprogramming level is sixteen. By the above definition, there are sixteen relations in the database (m=16). At a multiprogramming level of one, the Wisconsin multiuser benchmark will access one of the relations on behalf of a single user throughout the run repeatedly, whereas, with the new benchmark, every one of the relations has a uniform probability of access of 1/16. At a multiprogramming level of two, the Wisconsin multiuser benchmark will force each user to access its own private relation repeatedly throughout the run, while the new benchmark still maintains a 1/16 uniform probability of access to each relation by each user. Thus, while the physical size of the database (and, hence, the average seek distance for each page read/written) changes as a function of the multiprogramming level in the Wisconsin multiuser benchmark, the size of the database remains fixed in the new benchmark.

4.3.4. Query Types

In theory, all the relational algebra queries are in NC - i.e., they can be computed in poly-logarithmic time using polynomially many processors. This means that all relational algebra operators have the potential of being parallelized. It is beyond the scope of this dissertation to analyze the effect of the alternative partitioning strategies on all the relational algebra operators and the different algorithms proposed for each of these operators.

Instead of analyzing the performance of every single relational algebra operator with each of the alternative partitioning strategies, we elected to study the performance of different

selection queries. The major reason for concentrating on this operator is that it is one of the basic operators of any query plan and has a significant impact on the performance of a complex query. If a selection operator fails to feed tuples to the subsequent operators of a complex query at a sufficient rate, then the amount of parallelism that can be effectively applied by the subsequent operators will be limited. Furthermore, the alternative partitioning strategies directly affect the response time and throughput of selection queries, as the alternative partitioning strategies either localize the execution of selection queries to a subset of the processors (e.g., range) or employ all the processors hosting the fragments of the relation (e.g., round-robin).

To illustrate each of these, consider the join operator implemented via the hybrid hash join algorithm [DEWI84]. Assume that the size of the smaller relation participating in the join is smaller than the size of the aggregate main memory (no overflows). The query plan for this operator consists of two phases, termed the building and probing phases. During the building phase, the relation is scanned and a hash table is constructed on the value of the joining attribute. The probing phase consists of a scan on the second relation, hashing on the value of the joining attribute of each tuple of the second relation to see if there is a matching tuple in the hash table. Thus, this operator employs two scans. Since the scan operator determines a lower bound on the response time of the join operator, speeding up the execution time of the scan operator has the effect of reducing this lower bound. Furthermore, if the execution time of the scan operator during the probing phase is increased as the result of the partitioning strategy used for the probing relation, then the memory used for the hash table will be retained longer than absolutely necessary. Since memory, like CPU or I/O bandwidth, is a limited resource and retaining it for a longer interval than necessary can interfere with the execution of other queries which might have required that memory for their execution (e.g., some other concurrently executing join query).

For the purposes of this thesis, we classify the selection queries into: 1) queries that sequentially scan all the data pages of a relation to retrieve the qualifying tuples, and 2) queries that utilize an index structure to retrieve only the data pages that contain the

qualifying tuples. In Chapter 5, we demonstrate that a minimum number of processors must be used to execute the first class of queries because this query performs a series of sequential disk requests. If two or more of these queries are directed to a single processor, the two queries will almost certainly compete with one another for the disk drive, resulting in a series of random disk accesses. While sequential disk requests incur the rotational latency time of the disk drive, random disk requests incur both rotational latency and seek times, resulting in a higher response time and a correspondingly lower throughput. By localizing the execution of each concurrently executing query to a single processor, the probability of two or more queries being directed to different processors is increased. Thus, there is a higher probability of preserving the sequential nature of disk accesses made by each query, freeing it from interference with the other concurrently executing queries.

In the context of the second class of queries, we consider two distinct types of queries: 1) queries with low resource requirements, and 2) queries with high resource requirements. The reason for choosing these queries is to demonstrate the overheads associated with parallelism and the tradeoffs associated with localizing the execution of a query to a single processor. Chapter 5 will demonstrate that queries with low resource requirements should be directed to a single processor in order to avoid the overhead associated with parallelism. Otherwise, the overheads associated with parallelism will constitute a significant portion of the response time of each query. On the other hand, a large number of processors must be employed to execute queries with high resource requirements in order to: 1) speedup the execution time of each query, and 2) avoid the formation of hot spots and bottleneck processors in the presence of multiple concurrently executing queries.

In the next section, we describe the queries that were chosen to represent each of these classes of queries. In addition, we present the performance of a single processor instantiation of Gamma in a single user environment in order to provide further insight into the execution paradigm of the different selection queries.

Selection Queries

In this section, we motivate the specific selection queries that are used to evaluate the performance of the alternative declustering strategies. To make the discussion more concrete, the single-user performance of a single processor instantiation of Gamma with 8 KBYTE disk pages and a database consisting of ten relations is also presented. Each relation is based on the standard Wisconsin Benchmark relations [BITT83]. The tuples of each relation consist of thirteen 4-byte integer attributes and three 52-byte string attributes. Thus, each tuple is 208 bytes long. The cardinality of each relation is 41,700 tuples. We chose this relation because the performance of the alternative partitioning strategies will be analyzed using a 1,000,000 tuple relation[3]. Since a 24 processor instantiation of Gamma will be used to evaluate the performance of the alternative partitioning strategies, and assuming a uniform distribution of tuples across the processors, each processor will contain approximately 41,700 tuples. Thus, we can accurately compare the performance of each alternative partitioning strategy on a 24 processor configuration with the performance of a single processor for a given workload, and analyze the impact of each partitioning strategy individually (e.g., analyze whether the performance of the system with a given partitioning strategy increased by a factor of twenty four, and if not, the reasons for it).

We chose a 1,000,000 tuple relation for this performance evaluation because it provides enough work at each processor to cause the noise (e.g., retransmission of messages, disk head positioning, etc.) in the system to become insignificant. In [DEWI90], we studied the performance of different selection and join queries for a 100,000, 1,000,000 and a 10,000,000 tuple relation on a 32 processor instantiation of Gamma. In this study, we observed that with a 100,000 tuple relation, so few pages were processed by each processor that the overhead (operator initiation, termination, etc.,) became significant. Consequently, the response time of a

---

[3]The following paragraph will justify the use of a 1,000,000 tuple relation for this performance evaluation.

query varied significantly from one execution to the next. On the other hand, the performance of the system with 1,000,000 and 10,000,000 tuple relations provided enough work at each processor to render the overhead insignificant. We did not choose the 10,000,000 tuple relation because it would significantly increase the execution time of the different selection queries without providing additional insight into the performance of the alternative declustering strategies. In addition, it was not possible to store 10 copies of this relation on a 24 processor instantiation of Gamma (the cumulative storage capacity of 24 disk drives is only 7.92 Gigabytes, while the size of such a database is 20.8 Gigabytes).

All the response times and utilization numbers presented in this section were taken using a single processor instantiation of Gamma in a single-user environment with a low degree of data sharing. The response times presented in this section achieved statistical significance[4]. In addition, we present the CPU and disk utilization measurements to provide further insight into the performance of the system. The disk utilization was measured using a sampling method. With this method, the disk drive is sampled every 50 milliseconds to determine whether it is idle or busy servicing a request. Due to the large interval of time between the samples, there is a high margin of error associated with the disk utilization measurements presented in this thesis. On the other hand, the reported CPU utilization measurements are very accurate. The CPU utilization was measured by spawning a background process that increments a global variable in a loop. This process executes only when the CPU is idle (by assigning the lowest priority level to the process). For a well defined interval of time, we measured the number of times this global variable was incremented when the CPU was idle. Using this information and the value of the global variable during the execution of a benchmark, we can determine (modulo context switches) the fraction of the total execution time that the CPU was busy (i.e., CPU utilization).

---

[4]Refer to Section 4.3.1 for further discussion.

In general, the selection operator must interact with two devices, a disk and the interconnection network. Any time spent waiting on service from either device is wasted time. As a solution, Gamma utilizes a read-ahead process whenever a relation is accessed sequentially. The read-ahead process schedules the retrieval of the next data page from the disk while the selection operator is processing the current data page. The read-ahead process cannot be utilized when a non-clustered index is employed since there is no way of predicting the next data page required by the selection operator.

When the relation being accessed is stored as a heap, the selection operator must sequentially retrieve and process all the data pages of the relation. Table 4.1 presents the performance of a single tuple retrieval and two range selection queries in a single-user environment. As the selectivity factor of the range queries is increased, more tuples are sent to the application program, resulting in extra CPU processing (to create the communication packets and to place them onto the network). Therefore, as the selectivity factor of this query is increased, the CPU utilization of the system also increases (see Table 4.1). Since a constant number of disk pages are processed by each query, the average CPU service time per page also increases. Another observation from Table 4.1 is that the sum of the CPU and disk utilizations for all the queries using a sequential scan is greater than 100% as the read-ahead process overlaps the CPU processing of one page with the seek and transfer of the subsequent page.

Table 4.1
Sequential Scan of a 41,700 Tuple Relation on a Single Gamma Site

	AVG Response Time	AVG CPU Util (%)	AVG CPU-Time per page	AVG Disk Util(%)	AVG Disk-Time per page
Single Tuple	9.09 sec	93.70	7.35 msec	98.21	7.70 msec
1% Selection	9.11 sec	93.73	7.37 msec	96.70	7.60 msec
10% Selection	9.23 sec	94.32	7.50 msec	89.90	7.16 msec

While contradictory at first glance, for each query in Table 4.1, the CPU and disk service times per page are almost identical. There are two major reasons for this. First, the Intel processor does not have a DMA (Direct Memory Access module). Consequently, the CPU must transfer the data from a 2 KBYTE FIFO in the disk interface into the buffer pool. Second, and more significantly, this query performs a series of sequential disk accesses; thus, it utilizes the SCSI cache effectively to significantly reduce the disk service time. When the query issues a disk request for the first 8K block of data, the disk drive services this request and proceeds to read the next consecutive 37 Kilobytes of data into the SCSI cache (the capacity of the SCSI cache is 45 Kilobytes). When the query requests the next consecutive 8K disk page, the data is transferred from the SCSI cache without incurring the rotational latency time of the disk drive - i.e., the disk service time is reduced to the transfer time from the SCSI cache to the RAM. The SCSI cache speeds up the response time of this query by almost a factor of two[5].

For the new benchmark's sequential scan queries, we elected to use the 1% selection query to evaluate the performance of the alternative declustering strategies since its response time and resource requirements are almost identical to that of the other two queries.

For range selections queries using a B-tree access method, Gamma first traverses the B-tree to locate the upper and lower limits of the query with respect to the index records in the sequence set [COME79] of the B-tree (the B-tree leaf pages). All the data records that correspond to the index records between these two limits satisfy the selection query predicate[6]. Next, the selection operator distinguishes between the clustered index and the non-clustered index access methods. With a clustered index access method, the order of the values of the index records in the sequence set of the B-tree is the same as the order of the data records in the relation. The selection operator utilizes this information and performs a sequential scan of

---

[5] We determined this by disabling the SCSI cache and measuring the response time of the query.

[6] This strategy is also used for selection queries with an equality predicate since currently Gamma does not maintain any information about the uniqueness of attribute values.

the data records starting from the data record corresponding to the lower limit of the query to the data record corresponding to the upper limit of the query.

Table 4.2 presents the performance of the selection queries using a clustered index structure. The average disk transfer time per page for the single tuple retrieval query is higher than for the 1% and 10% range selection queries because all of its disk requests are random. This query performs two random disk accesses to traverse the depth of the B-tree and locate the appropriate index record in the leaf page of the B-tree and a final random disk request to retrieve the desired data record. For the 1% and 10% selection queries, there are two random disk accesses to traverse the depth of the tree (to determine the data records corresponding to the lower and upper limits of the query) and another random disk access to the data record corresponding to the lower limit of the query. Next, a sequential scan of the data pages starting from the physical record corresponding to the lower limit of the query to the physical record corresponding to the upper limit of the query is performed. Therefore, all the disk accesses made by the single tuple retrieval queries are random while the range queries result in a series of random and sequential disk accesses. Random disk requests cannot take advantage of the SCSI cache. Furthermore, they result in longer seek times than the sequential disk requests. In addition, the range queries utilize the read-ahead process for the sequential retrieval of the data pages, resulting in the sum of the CPU and disk utilizations being greater than 100% once again.

Table 4.2
Scan of a 41,700 Tuple Relation
Using a Clustered Index on a Single Gamma Site

	AVG Response Time	AVG CPU Util(%)	AVG CPU-Time per page	AVG Disk Util(%)	AVG Disk-Time per page
Single Tuple	0.12 sec	74.42	12.33 msec	43.98	27.39 msec
1% Selection	0.26 sec	83.18	11.83 msec	57.28	10.68 msec
10% Selection	1.16 sec	95.22	9.03 msec	59.78	5.93 msec

In Table 4.2, the average CPU service time per page for the single tuple retrieval query is higher than that of the range queries. We speculate that this is because the CPU processing time of an index page is higher than that of a data page. There are two major reasons for this speculation. First, for each index page, a binary search of the index records must be performed in order to determine either what index page to follow or which data page to retrieve. Second, the number of index records per B-tree page is significantly higher than that of a data page. While each query processes the same number of index pages, the range queries process a larger number of data pages. Thus, for the range queries, the higher CPU service time of the index pages is amortized over a larger number of pages (index + data). This explanation also applies to the lower CPU processing time per page for the 10% selection query when compared with the 1% selection query.

With a non-clustered index, since the order of the values of the index records is not the same as the order of the data records in the relation, the selection operator must traverse the index records between the lower and upper limit and retrieve the data record corresponding to each index record. The result is approximately one random I/O for each tuple retrieved [YAO79]. In addition, since almost all the accesses are random, neither the SCSI cache nor the read-ahead process can be utilized.

Table 4.3 presents the performance of the single tuple retrieval and 1% selection queries using a non-clustered index. The measurements for the 10% selection are not presented since the execution of this query via a sequential scan is faster than using a non-clustered index.

Table 4.3
Scan of a 41,700 Tuple Relation
Using a Non-Clustered Index on a Single Gamma Site

	AVG Response Time	AVG CPU Util(%)	AVG CPU-Time per page	AVG Disk Util(%)	AVG Disk-Time per page
Single Tuple	0.12 sec	74.42	12.33 msec	43.98	27.39 msec
1% Selection	9.76 sec	76.20	8.86 msec	100.00	27.08 msec

Therefore, our optimizer elects to ignore the non-clustered index for the 10% selection query. The performance of a single tuple retrieval using a non-clustered index is identical to that of a single tuple retrieval using a clustered index since both query types traverse the depth of the B-tree to locate the appropriate index record and then make a final random disk request to retrieve the appropriate data record. The performance of the 1% selection using a non-clustered index is I/O bound due to the large number of random disk operations.

As discussed in section 3.4 of Chapter 4, in the context of selection queries that use an access method, we wanted to consider two types of queries: 1) queries with high resource requirements, and 2) queries with low resource requirements. We choose the single tuple retrieval using a non-clustered index to represent queries with low resource requirements for the benchmark. The 10% selection using a clustered index was elected to represent queries with high resource requirements. We discarded the 1% selection using a nonclustered index from further consideration for two reasons. First, its performance is completely I/O bound. Second, when a relation is range partitioned, the execution of this query via a sequential scan is faster than using a non-clustered index. To illustrate, assume a 24 processor configuration with a 1,000,000 tuple relation horizontally declustered across all the processors using the range partitioning strategy. The range partitioning strategy attempts to localize the execution of a query regardless of its selectivity factor.[7] When this query is directed to a single processor, it becomes a 24% selection query here since it retrieves 1% of the tuples from a single fragment of the relation rather than retrieving $\frac{1}{24}$ of the tuples from each fragment of the relation. Thus, for the 1% selection query using the range partitioning strategy, our optimizer elects a sequential scan of the relation instead of utilizing the non-clustered index access method.

To summarize, in this thesis we will study the impact of the alternative partitioning strategies on the performance of selection queries. We will consider only selection queries which

---

[7]Assuming the query predicate is accessing the partitioning attribute.

retrieve their result tuples back to an application program. The selection queries to be used in our performance evaluation are: 1% selection via a sequential scan, 10% selection using a clustered index, and single tuple retrieval using a non-clustered index.

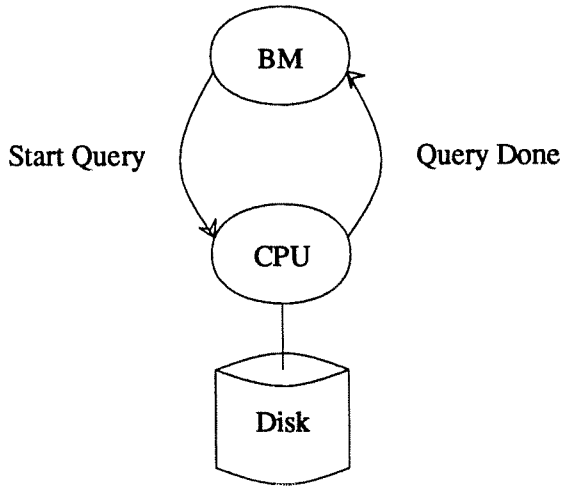
CHAPTER 5

EVALUATION OF ALTERNATIVE DECLUSTERING STRATEGIES

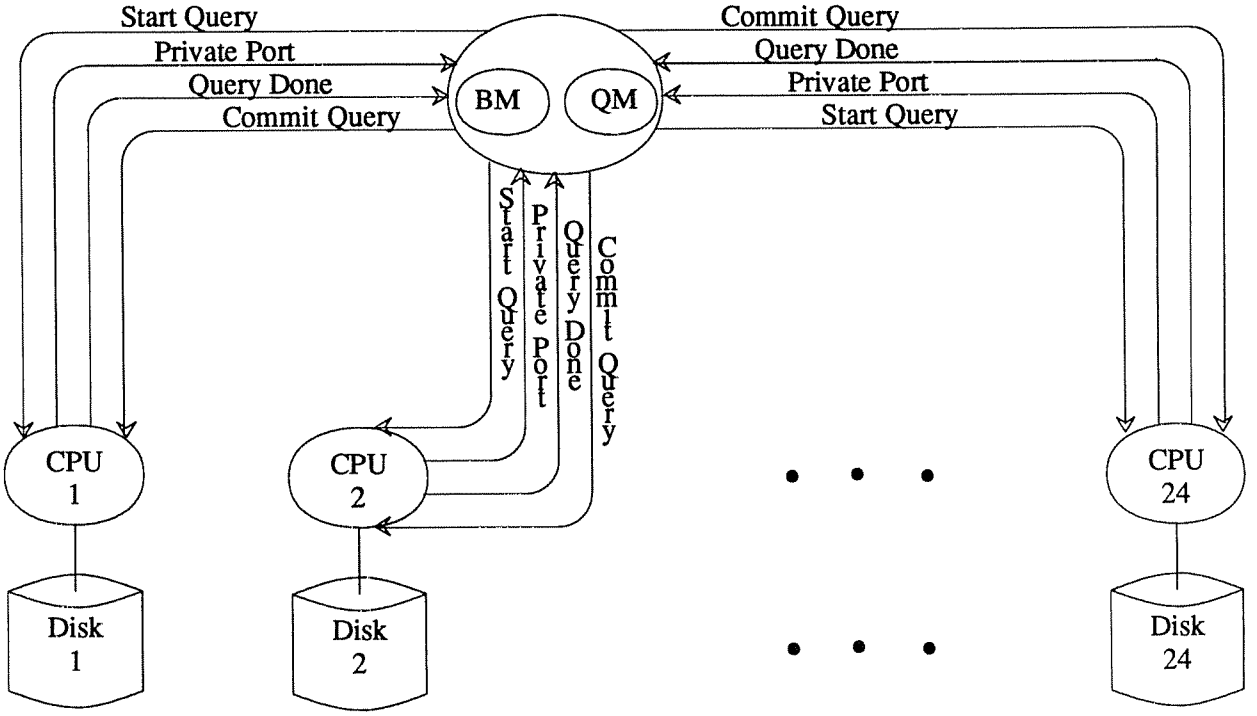
This chapter presents the performance of the different selection query types with the alternative declustering strategies using a twenty four processor instantiation of Gamma. For the purposes of this evaluation, the database consisted of ten relations (m = 10). The cardinality of each relation was 1,000,000 tuples. The characteristics of each relation were based on the standard Wisconsin Benchmark relations [BITT83]. The alternative declustering strategies distributed the tuples of each relation uniformly across the processors.

Before proceeding to the performance evaluation itself, we first must describe the execution paradigm for each of the alternative partitioning strategies. As shown in Figure 5.1.a, the benchmark (BM) process acts as a terminal emulator that generates and submits queries to Gamma for execution. For each new query, the BM process randomly selects one of the 10 relations in the database and randomly generates a predicate for the selection query (both using a uniform random number generator). As described in Chapter 1, the Gamma query optimizer utilizes the information provided by the alternative declustering strategies in order to limit the number of processors employed to execute certain queries.

As shown in Figure 5.1.b, when a query must be executed by multiple processors, it is first sent to a Query Manager (QM) process which assumes responsibility for its execution. The QM process sends the query to each processor that the query optimizer has indicated to execute the query. When each processor finishes executing the query, it sends a "query done" message back to the QM process. If the QM receives a "query done" message from each processor, it sends a "commit" message to each processor and closes all of the communication ports. Finally, it notifies the process that submitted the query of its successful execution. If, on the other hand, the query is aborted at a processor (generally because of a concurrency



a. Single Processor Query



b. Multi-Processor Query

Figure 5.1

control deadlock) that processor sends an "abort" message to the QM. The QM, in turn, sends an "abort" message to each participating processor and then notifies the process that submitted the query. On the other hand, queries that execute on only a single processor are sent directly to the proper processor for execution, bypassing the QM process (see Figure 5.1.a). As illustrated by Figure 5.1, significantly fewer messages are required to control the execution of a single processor query, and as will be demonstrated, this difference has a significant effect on the performance of the alternative partitioning strategies.

The benchmark process (BM) and the query manager process (QM) are two independent entities that are generally located on different processors, but they were placed on the same processor for these experiments. In order to simulate multiple concurrently executing users, one BM and one QM process was employed for each user. Eight processors were used for running the BM and QM processes in order to avoid a bottleneck from forming. For all the experiments presented below, the CPU utilization of each of these 8 processors was less than 100%.

In the rest of this chapter, we will present and discuss the response time and throughput of different selection query types with each of the alternative partitioning strategies. CPU and disk utilization measurements of the system are also presented to help clarify our explanations and to provide extra insight into the performance of each query. The performance of the system for the low degree of data sharing is presented as solid lines, while for the high degree of data sharing, it is presented as dash lines. The response times presented in this chapter achieved statistical significance[1].

When a query retrieves on a non-partitioning attribute, it is directed to all the processors and each processor searches its fragment of relation in parallel with the other processors. In this thesis, we term this a Non-Partitioning Attribute Selection Query (NPA_SQ) and characterize its performance as compared with that provided by the range and hash declustering strategies.

---

[1] See Chapter 4, Section 3.1.

5.1. 1% Selection Via a Heap Storage Structure

With a heap storage structure, all the data pages in the relation must be sequentially retrieved from the disk drive and processed. The response time and throughput of the alternative partitioning strategies for the 1% selection query via a heap storage structure is presented in Figures 5.2 and 5.3 respectively.

While with the range declustering strategy the optimizer has sufficient information to almost always localize the execution of this query to a single processor, with the hash partitioning strategy and in the case of the NPA_SQ all the processors are utilized to execute this query. (Recall that the hash declustering strategy cannot localize the execution of range queries[2].) The performance of the hash partitioning strategy and the NPA_SQ are presented as

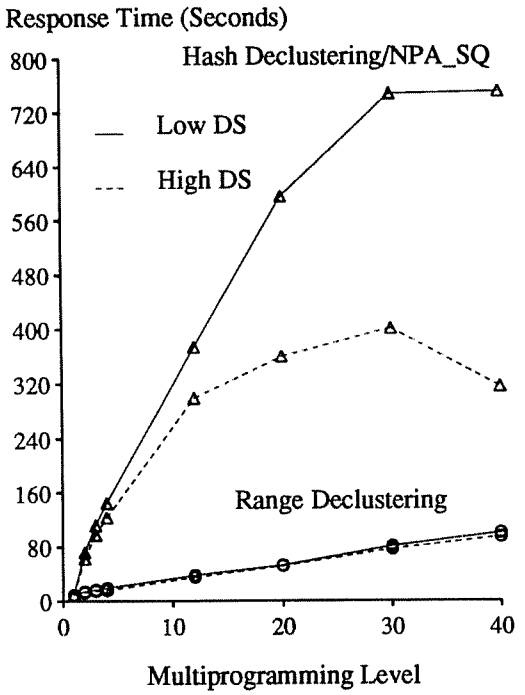


Figure 5.2: 1% Non-Indexed Selection

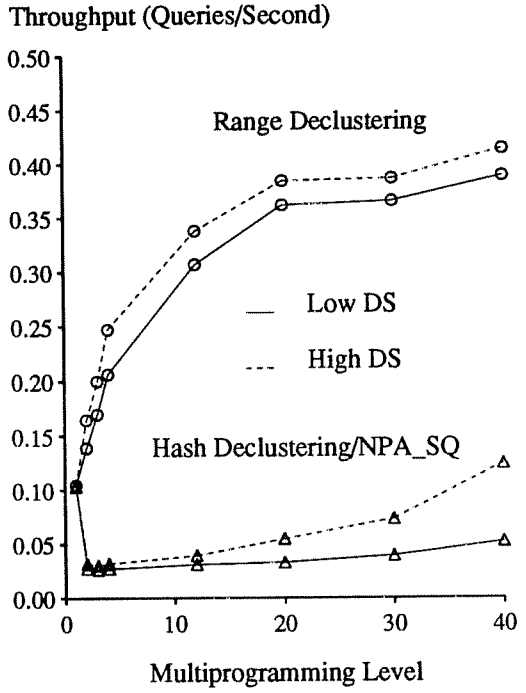


Figure 5.3: 1% Non-Indexed Selection

---

[2] Note that if this query was an exact match query, the performance of the hash and range declustering strategies would have been identical since the hash partitioning strategy can also localize the execution of exact match queries.

a single curve since the difference in their response time and throughput is statistically insignificant.

At a multiprogramming level of one, the throughput of the system is identical for each of the alternative declustering strategies as the execution time of the query on a single processor is high enough to render the overhead associated with a multiprocessor query insignificant.

For the hash declustering strategy and in the case of NPA_SQ, the throughput of the system decreases markedly from a multiprogramming level of 1 to 2. At a multiprogramming level of two, the two concurrently executing queries either access different relations (low degree of data sharing) or different parts of the same relation (high degree of data sharing). Thus, the two queries result in a series of random rather than sequential disk requests. Although at first glance this distinction may not seem sufficient to cause the large performance difference of Figure 5.3, it is important to realize that these random requests are performed for every single data page in the relation. Furthermore, random disk requests render the SCSI cache ineffective, and the SCSI cache can speed up the execution time of this query by almost a factor of two. Consequently, when it becomes ineffective, the performance of the system degrades significantly.

At higher multiprogramming levels, the throughput for the hash partitioning strategy begins to increase as the result of a higher percentage of buffer pool hits (see Figure 5.4). Since the database consists of 10 relations, at high multiprogramming levels several concurrently executing queries might access the same relation and become "synchronized"[3] with one another to result in a higher percentage of buffer pool hits. A second reason for the increase in throughput is that the disk controller utilizes an elevator algorithm. By utilizing this algorithm, the distance traveled by the head of the disk decreases for several (more than two) pending I/O requests, resulting in a lower average seek time.

---

[3] A set of queries is considered "synchronized" if their access to the data pages of a relation occur close enough in time that they can share each others' data pages in the buffer pool.

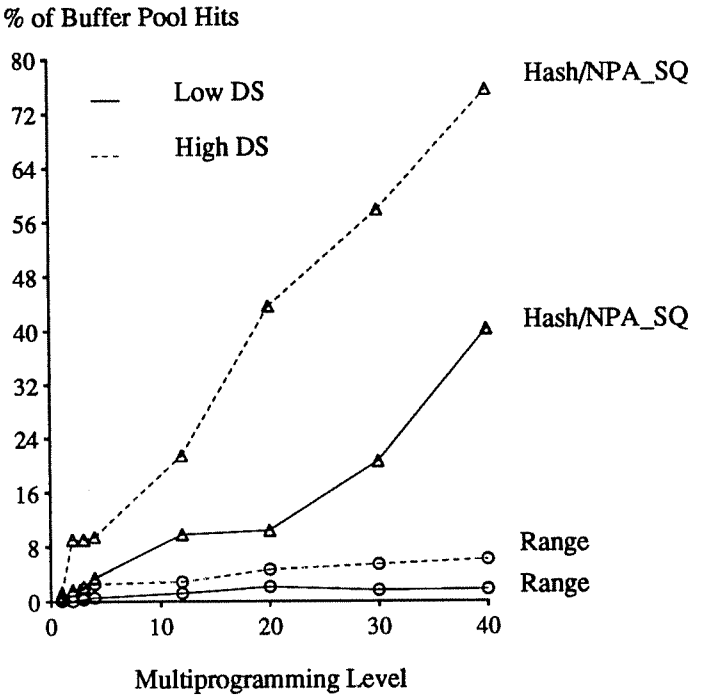


Figure 5.4: 1% Non-Indexed Selection

With the range declustering strategy, the throughput of the system increases at multiprogramming levels higher than one since idle resources become utilized by the additional concurrently executing queries. The range declustering strategy utilizes the SCSI cache most effectively since it can almost always direct two concurrently executing queries to different processors, avoiding any interference that might render the SCSI cache ineffective. However, the throughput of the system does not increase linearly from a multiprogramming level of one to two because the attribute value of the tuples retrieved by each query is generated using a uniform random number generator. While the random number generator results in a uniform distribution of all the submitted queries across all the processors for the complete execution of the benchmark, at any instant in time, it does not guarantee an identical number of queries executing concurrently at each processor. To illustrate this point, in Table 5.1 the percentage of queries executed at each processor and the number of queries executed concurrently at each processor at a multiprogramming level of two is presented. The results reveal that for the complete execution of the benchmark, each processor executed approximately the same

Table 5.1
Distribution of Workload Across the Processors (MPL = 2)

Processor Number	% of Total Queries Executed	% of Time Executing 0 Queries	% of Time Executing 1 Query	% of Time Executing 2 Queries
1	4.19%	98.01%	1.94%	0.05%
2	4.11%	97.78%	2.02%	0.20%
3	4.16%	96.81%	2.02%	1.17%
4	4.22%	97.69%	1.68%	0.63%
5	4.14%	95.89%	2.26%	1.85%
6	4.22%	97.57%	1.88%	0.55%
7	4.16%	98.03%	1.43%	0.54%
8	3.98%	96.80%	2.07%	1.13%
9	4.20%	95.32%	2.00%	2.68%
10	4.22%	97.33%	1.90%	0.77%
11	4.17%	94.86%	2.21%	2.93%
12	3.98%	96.60%	2.12%	1.28%
13	4.24%	97.14%	1.73%	1.13%
14	4.11%	96.50%	1.99%	1.51%
15	4.18%	94.58%	2.89%	2.52%
16	4.22%	97.88%	2.08%	0.05%
17	4.24%	97.09%	1.55%	1.37%
18	4.24%	98.21%	1.79%	0.00%
19	4.21%	96.51%	1.99%	1.50%
20	4.20%	96.95%	2.23%	0.83%
21	4.20%	96.30%	1.90%	1.80%
22	4.10%	97.07%	2.46%	0.47%
23	4.18%	96.91%	2.52%	0.57%
24	4.14%	97.88%	2.12%	0.01%

percentage of queries (approximately 4.2%). The results also indicate that some of the processors executed two queries concurrently some of the time. If the two concurrently executing queries had been distributed uniformly across the 24 processors at every instant in time, then a single processor should have never observed two concurrently executing queries. Thus, the throughput of the system cannot increase linearly since, at any instant in time, a processor might execute more than its fair share of concurrently executing queries while other processors are idle waiting for work.

The percentage of buffer pool hits for the range declustering strategy is significantly lower than that of the hash declustering strategy (see Figure 5.4). This is because with the range partitioning strategy, a new dimension of randomness is introduced into the experiments. Not only do the newly created queries randomly choose a relation to access, but they also randomly

choose a processor to execute on. (This occurs because the range declustering strategy can almost always direct each query to a single processor.) Since the random nature of the workload results in an unequal number of concurrently executing queries at each processor, the load characteristics of each processor is slightly different. Consequently, the probability of a set of queries becoming "synchronized" and capable of sharing data pages in the buffer pool is very low.

The above discussion is best clarified using an illustration. Assume that 10 queries are executing concurrently on processor number one. Furthermore, assume that all 10 queries are "synchronized" and share data pages in the buffer pool. Once these queries commit, their corresponding terminals must create and submit new queries (a new query iteration). Each new query retrieves 10,000 tuples based on attribute values generated using a random number generator. Thus, each query is directed to one of the 24 processors based on the randomly selected attribute values. Assume that logical processors 1 through 6 are each now assigned a single query, while processors 7 and 8 are each assigned two queries. Therefore, the load characteristics of processors 7 and 8 will be different than the other six processors. Furthermore, since the two queries on processors 7 and 8 have a longer execution times, they are no longer "synchronized" with the other concurrently executing queries. When the queries executing on processors 1 thorough 6 commit, their terminals must create and submit new queries. The response time of these six queries will be identical since the characteristics of each processor (CPU power, disk transfer rate, etc.,) and the amount of work performed at each processor is identical. Therefore, the six new queries are "synchronized" and can share data pages if directed to a single processor (probability of $(\frac{1}{24})^6$). If any of these new queries are directed to processors 7 or 8, they will not be "synchronized" with the two already executing queries. Furthermore, they will no longer be "synchronized" with the other five executing queries. When the queries on processor 7 and 8 commit, their corresponding terminals must create and submit new queries. The two new queries will not be "synchronized" with the other eight concurrently executing queries. Furthermore, the probability of these two new queries being

directed to the same processor is very low ($\frac{1}{576}$). Thus, after a short interval of time, a set of "synchronized" queries becomes completely "unsynchronized" due to the random nature in which queries choose a processor to execute on.

In conclusion, for a sequential scan query, the best throughput is obtained by the declustering strategy that localizes the execution of this query to a single processor since the best performance is obtained when the query executes all by itself at a single processor and performs a series of sequential disk requests. By localizing the execution of the query to a single processor, there is a higher probability of maintaining the sequential nature of disk requests made by a query, free from interference of the other concurrently executing queries. Thus, for the sequential scan queries, the optimal partitioning strategy is the range partitioning strategy.

5.2. 10% Selection Using a Clustered Index Structure

With a clustered index, the order of the values of the index records in the sequence set of the B-tree is the same as the order of the data records in the relation. Two types of disk requests are made by range selection queries that use a clustered index structure: random and sequential. The traversal of the B-tree to locate the upper and lower limits of the query with respect to the actual data records is random, while the retrieval of data records between the lower and upper limit markings is sequential.

Figure 5.6 presents the throughput of the alternative declustering strategies for a 10% selection query using a clustered index. We have included only results for a low degree of data sharing as the performance of the system is almost identical with a high degree of data sharing. With the hash partitioning strategy and in the case of NPA_SQ, all the processors must participate in the execution of a range query regardless of which attribute the selection predicate is applied to. Therefore, the execution paradigm of the 10% selection query is identical for both the hash partitioning strategy and the NPA_SQ execution paradigm and is presented as a single curve rather than two overlapping curves. The throughput of the system with these two partitioning strategies almost doubles from a multiprogramming level of one to two. This is

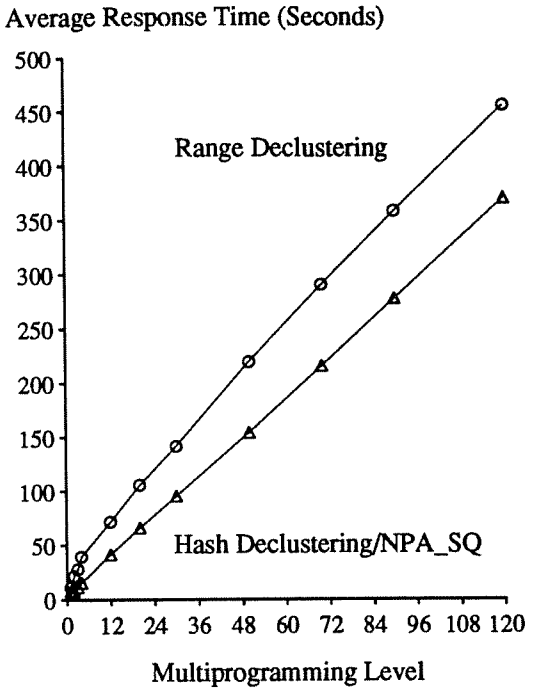


Figure 5.5: 10% Indexed Selection

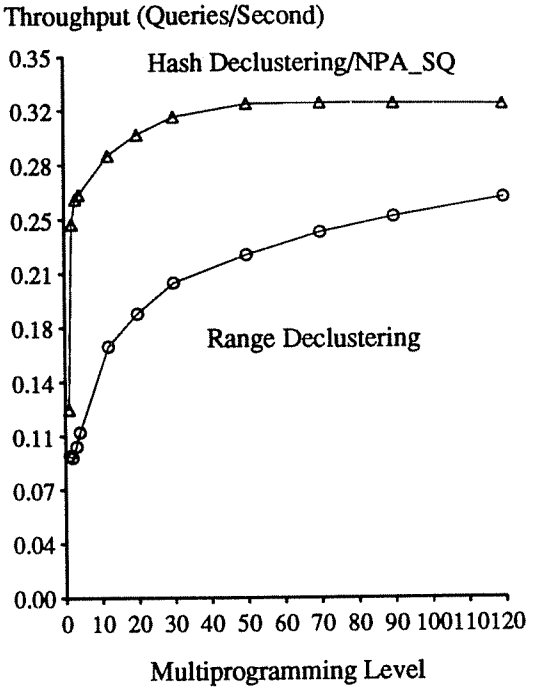


Figure 5.6: 10% Indexed Selection

because the benchmark processor is a bottleneck at a multiprogramming level of one. At this multiprogramming level, when the BM process submits a query, all the 24 processors transmit their resulting tuples almost simultaneously to the BM process. The BM process does not generate a new query until all the resulting tuples of the current query are processed. During the time that the BM process is consuming these tuples, the 24 processors are idle waiting for work. Consequently, as the multiprogramming level is increased from one to two, the throughput of the system doubles as the idle resources are utilized by the additional concurrently executing query.

In a separate experiment, we eliminated the BM processor as the bottleneck by forcing each processor to drop the qualifying tuples when executing a query - i.e., not transmit its output tuples to the BM process. Under this set of circumstances, the throughput of the system was 0.63 queries/second at a multiprogramming level of one. However, at multiprogramming levels higher than one, the throughput of the system was identical to that of Figure 5.6 since the BM processor is not a bottleneck at those multiprogramming levels. Observe that for this

experiment, the throughput of the system decreases from a multiprogramming level of one to two (i.e., from 0.63 to 0.25 queries/second) because the two queries result in random disk requests at each processor rendering the SCSI cache ineffective.

Since the range partitioning strategy localizes the execution of all range queries on the partitioning attribute, it directs the 10% selection query to two or three processors. At a multiprogramming level of one, the hash partitioning strategy outperforms the range declustering strategy because it utilizes parallelism effectively to perform $\frac{1}{24}$ of the work at each processor while the range declustering strategy performs majority of the work in a sequential manner.

For the range declustering strategy, when the execution of this query is localized to two or three processors, it sequentially retrieves and processes a very large number of data pages. Thus, the SCSI cache has a significant impact on the performance of this query. Every time two of these queries collide and execute concurrently on the same processor, the SCSI cache is rendered ineffective. Although the probability of a collision is fairly low, its impact on the performance is significant enough to cause the overall throughput of the system to decrease from a multiprogramming level of one to two.

At multiprogramming levels higher than two, the throughput of the system begins to increase as a larger number of idle processors are utilized by the additional concurrently executing queries. However, at high multiprogramming levels, one would expect the throughput of the two partitioning strategies to converge since the processing capability of the system and the amount of work performed by this query is well defined. This does not happen for two major reasons. First, as discussed before, the random nature of the workload does not guarantee a uniform distribution of the concurrently executing queries among the processors for the range declustering strategy. Consequently, at high multiprogramming levels, certain processors become a bottleneck as they execute a larger than their fair share of concurrently executing queries while other processors are idle waiting for work.

Second, the range declustering strategy cannot utilize all the processors to their maximum capacity. In order to explain this, consider all the possible ways that three processors

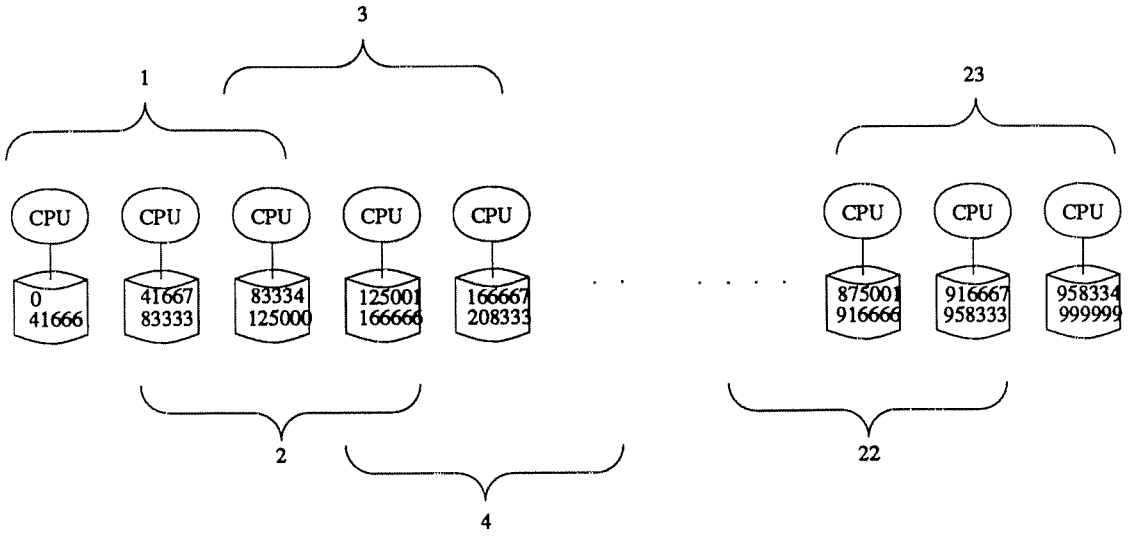


Figure 5.7

can be employed to execute a 10% range selection query. As illustrated by Figure 5.7, there are only 23 possible ways for a range query to overlap three processors. Observe that the two processors containing the upper and lower limits of the partitioning attribute values (processors 1 and 24) can participate in only one combination, processors 2 and 23 can participate in two combinations, while the remaining processors can participate in three combinations. Therefore, at all multiprogramming levels, processors 1, 2, 23, and 24 have a lower probability of being accessed. In Figure 5.8, the CPU utilization of the individual processors at a multiprogramming level of 120 is presented. The results provide further evidence that these processors were not utilized as much as the other processors. Thus, the range partitioning strategy can fully utilize only 20 out of the 24 processors while the hash declustering strategy can utilize all the 24 processors.

A solution to this problem is to assign the lower and upper limit ranges of the different relations to different processors (in our current layout, the upper and lower limits of all ten relations are assigned to the same processor). This solution is effective when the relations are accessed uniformly. However, for a skewed distribution of access, such an assignment of ranges will not solve this limitation of the range declustering strategy. The multidimensional

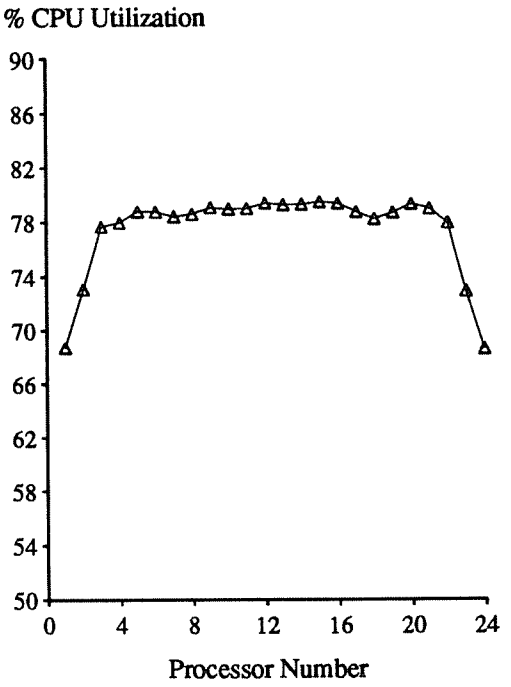


Figure 5.8: CPU Utilization at MPL = 120

declustering strategy, presented in Chapter 6, provides a solution for this limitation.

5.3. Single Tuple Retrieval Using a Non-clustered Index Structure

Figure 5.9 presents the throughput of Gamma for a single tuple retrieval query using a non-clustered index. In the case of a NPA_SQ, all 24 processors are employed to execute this query. Consequently, it incurs the overhead associated with controlling the execution of the query on multiple processors (see Figure 5.1). This overhead causes some resources to remain idle at a multiprogramming level of one. Thus, as the multiprogramming level is increases from one to four, the throughput increases linearly as the resources are utilized without interference to the currently executing queries. In the case of NPA_SQ, only one of the 24 processors finds the specified tuple, the remaining 23 processors search their fragment of relation only to find no qualifying tuples. Consequently, at a multiprogramming level of 20, the CPU of each processor becomes 100% utilized and the throughput of the system levels off.

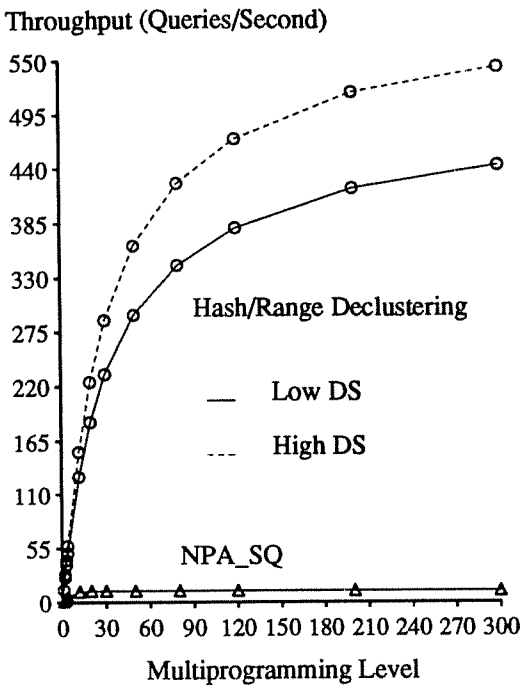


Figure 5.9: Single Tuple Indexed Selection

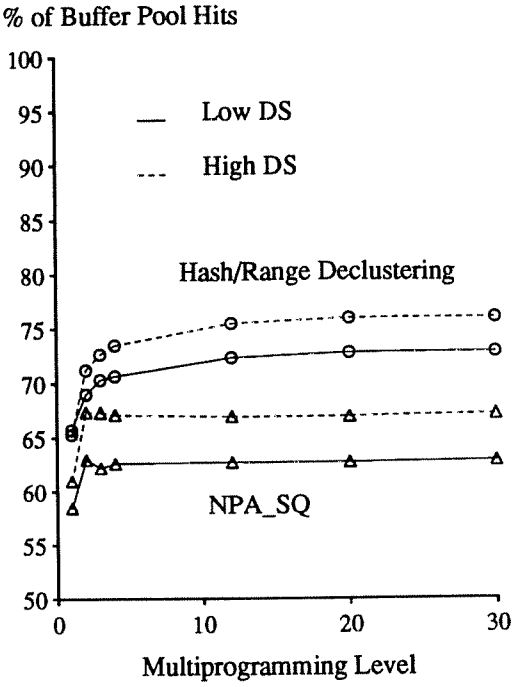


Figure 5.10: Single Tuple Indexed Selection

The range and hash declustering strategies direct the execution of this query to a single processor and avoid the overhead associated with controlling the execution of the query on multiple processors. Thus, they outperform the execution paradigm of NPA_SQ by almost a factor of 10 at a multiprogramming level of one. At high multiprogramming levels, they outperform the NPA-SQ by almost a factor of 46 because they direct each submitted query to the processor with the relevant tuple, freeing other processors to execute some other concurrently executing query (rather than requiring other processors to search their fragment of the relation for each submitted query only to find no qualifying tuples).

With the range and hash partitioning strategies, the throughput of the system as a function of the multiprogramming level increases sharply because the idle resources are utilized by the additional concurrently executing queries. However, the throughput does not increase linearly since the random nature of the workload cannot guarantee the same number of concurrently executing queries at each processor at any instant in time. In [GHAN90a], we demonstrated how one of the processors can become a permanent bottleneck for the multipro-

cessor database machine at very high multiprogramming levels. In the experiments presented here, we could not drive the multiprogramming level of the system high enough to cause one of the processors to become a permanent bottleneck. This is because the benchmark processors ran out of memory at very high multiprogramming levels. We refer the interested reader to [GHAN90a] for a detailed explanation of the bottleneck phenomena and its impact on the throughput of the system.

With the range and hash declustering strategies, the throughput of the system is significantly higher for the high degree of data sharing than for the low degree of data sharing primarily because of a higher percentage of buffer pool hits (see Figure 5.10). In the case of the NPA_SQ, the higher percentage of buffer pool hits provided for the high degree of data sharing has no impact since the overhead of controlling the execution of the query renders these savings insignificant.

In conclusion, if the resource requirements of a query is very low, it is advantageous to direct the execution of the query to the processor with the relevant fragment of the relation in order to avoid the overhead associated with parallelism.

5.4. Summary

The results presented in this chapter reveal that for a shared-nothing multiprocessor database machine, no partitioning strategy is superior under all circumstances. Rather, each partitioning strategy outperforms the others for certain query types. The major reason for this is that there exists a tradeoff between exploiting intra-query parallelism by distributing the work performed by a query across multiple processors and the overhead associated with controlling the execution of a multiprocessor query. Localizing the execution of queries that retrieve a few tuples and have minimal resource requirements results in the best system response time and throughput. The best response time is obtained because the overhead associated with parallelism is minimized. The throughput is maximized because the processors that do not contain any relevant tuples are freed to execute some other concurrently executing query. On the other hand, for queries with high resource requirements, certain

tradeoffs are involved. For instance, with the 10% selection using a clustered index, we observed that the throughput with the hash partitioning strategy is higher than with the range partitioning strategy. In general, with access methods that result in the retrieval of only the relevant tuples from the disk, if the resource requirements of the query are very low, it is advantageous to localize the execution of the query to a single processor. For queries with high resource requirements, a high degree of parallelism must be employed in order to: 1) reduce the response time of each query, and 2) avoid the formation of hot spots and bottleneck processors (i.e., to achieve good load balancing). While the hash partitioning strategy localizes the execution of the exact match selection queries (e.g., a single tuple retrieval using a non-clustered index), the range partitioning strategy attempts to localize the execution of all queries regardless of their selectivity factor. At the other end of the spectrum, when a query accesses a non-partitioning attribute, it is directed to all the processors containing the fragments of the referenced relation.

For sequential scan (i.e., non-indexed) queries, the best response time and throughput is obviously achieved by localizing the execution of each query to a single processor. The system generally performs best when the query executes all by itself at a single processor and performs a series of sequential disk requests. By localizing the execution of the query to a single processor containing the fragment with the relevant tuple, there is a higher probability of maintaining the sequential nature of disk requests made by a query, free from interference of the other concurrently executing queries. Thus, for the sequential scan queries, the optimal partitioning strategy is the range declustering strategy.

In a multiprocessor configuration with the range declustering strategy, some processors are utilized more than others for the range queries that generally overlap the range of values of two or more adjacent processors (e.g., 10% selection query using a clustered index). The processors corresponding to the upper and lower limits of the range of the partitioning attribute values do not participate in the execution of as many queries as the other processors in the environment. This is a consequence of how the range partitioning strategy partitions a relation across the processors.

A simple solution to the above problem is to analyze the queries accessing a relation and determine the percentage of time the range of the queries overlap two or more processors. Using this information, more tuples can be assigned to the fragments corresponding to the upper and lower limits of the partitioning attribute value of the relation. By assigning more tuples to these fragments, the probability of access to the processors containing the upper and lower limits of the query is increased. This is not an optimal solution since it might degrade the performance of other query types accessing the relation. To illustrate, assume a query that retrieves one tuple via a sequential scan using a non-partitioning attribute. The response time of this query will be higher with the non-uniform distribution of tuples across the processors because the query is directed to all the processors and the processors corresponding to the upper and lower limits of the partitioning attribute value will have to process more tuples than the other processors. The **multidimensional partitioning** strategy presented in the next chapter provides a better solution for this limitation.

It is important to remember that all the results and conclusions presented in this chapter are contingent on the following assumptions: 1) a uniform distribution of tuples of a relation across the processors, 2) a uniform distribution of access to the tuples of a relation, and 3) a uniform distribution of the partitioning attribute value.

CHAPTER 6

THE MULTIDIMENSIONAL PARTITIONING STRATEGY

In this chapter, we present the **multidimensional** partitioning strategy (MDPS). The MDPS differs from the range and hash declustering strategies in two ways. First, it declusters a relation by utilizing the characteristics of the queries that access the relation. As demonstrated in Chapter 5, the resource requirements of a query determine if the query should be directed to a single processor or multiple processors. By using this information, the MDPS declusters a relation with the primary objective of providing the appropriate degree of intra-operator parallelism for the selection operators that access the relation.

Second, it declusters a relation based on several attributes rather than a single attribute. Consequently, it can localize the execution of a greater variety of queries. In order to clarify this difference, recall the STOCK relation from Chapter 1. Assume that 50% of the queries (termed query type A) access the relation using an equality predicate on the *ticker_symbol* attribute (e.g., retrieve STOCK.all where STOCK.ticker_symbol = AXP). The other 50% of the queries (termed query type B) use a range predicate on the *price* attribute (e.g., retrieve STOCK.all where STOCK.price > 10 and STOCK.price <= 20). Furthermore, assume that query type B retrieves and processes only a few tuples. With this workload definition, the appropriate access methods are a non-clustered (or hash) index on the *ticker_symbol* attribute and a clustered index on the *price* attribute of the STOCK relation. However, either attribute qualifies as the declustering attribute since both queries have minimal resource requirements and should be directed to a single processor. The range and hash partitioning strategies can decluster a relation based on only a single attribute. Consequently, each will direct 50% of queries in the workload to all the processors and hence incur the overhead associated with using more processors than absolutely necessary. The MDPS is able to decluster the STOCK

relation using both the *ticker_symbol* and *price* attributes. Consequently, it can localize the execution of both queries.

As illustrated by this example, an assumption of the MDPS is that the resource requirement and frequency of occurrence of each query accessing the relation is defined. Furthermore, the MDPS is designed for queries with low resource requirements whose execution should be directed to a small fraction of the processors available. If the resource requirements of a class of queries accessing a given attribute is high then that attribute should not be used as a declustering attribute in order to ensure that all processors are used to execute it. But, what fraction of processors should be employed by the queries accessing an attribute in order to qualify that attribute as one of the partitioning attributes? Defining such a threshold is a hard problem and part of the future work of this dissertation. However, from the results presented in Chapter 5, if the queries accessing an attribute should be directed to only one or two processors, one can safely conclude that the attribute should be used as one of the partitioning attributes.

The MDPS can be compared with the range and hash declustering strategies by analyzing the number of processors employed by each declustering strategy. The declustering strategy that most closely approximates the ideal degree of intra-query parallelism (computed based on the resource requirements of the queries accessing the relation) is a superior declustering strategy because by using the appropriate number of processors: 1) the response time of the queries is improved since they do not incur the overhead associated with controlling their execution on more processors than absolutely necessary, and 2) the throughput of the system is increased as the processors that do not contain the relevant fragments are freed to execute other queries.

While the range partitioning strategy assigns a range of values of a single attribute to each processor, the MDPS assigns a range of values of several partitioning attributes to each processor in the system. Several existing multi-attribute access methods could be used as the basis of the MDPS: R tree [GUTT81], R+ tree [SELL87], Grid File Structure[NIEV84], k-d-b tree

[ROBI81], etc. We believe that the Grid file structure is the most appropriate multi-attribute structure for the MDPS because unlike the trees constructed by the other access methods, it constructs a grid directory that maintains a straightforward (one-to-one) correspondence between the elements of the directory and the logical record space of multiple declustering attributes. This allows the MDPS to easily manipulate the directory of the Grid file structure to ensure that the appropriate number of processors are used to execute the queries accessing each of the different attributes. Before explaining the MDPS algorithm for constructing a grid directory on a relation, we present an example both to illustrate how a grid file directory is used by the MDPS and to further motivate the goals of the MDPS.

Example 1:

Recall the STOCK relation and the two queries (A and B) that access this relation. Assume a system consisting of nine processors (numbered from 1 to 9). Figure 6.1 presents a two dimensional grid file directory on the STOCK relation and a mapping of the elements of the grid directory to the processors. Each element in this directory corresponds to a fragment of the relation. The rows of the directory correspond to the range of values for the *price* attribute, while the columns correspond to the range of values for the *ticker_symbol* attribute. The grid file

Ticker_Symbol

Price	*A-D*	*E-H*	*I-L*	*M-P*	*Q-T*	*U-Z*
0 - 10	*1*	*1*	*4*	*4*	*7*	*7*
11 - 20	*1*	*1*	*4*	*4*	*7*	*7*
21 - 30	*2*	*2*	*5*	*5*	*8*	*8*
31 - 40	*2*	*2*	*5*	*5*	*8*	*8*
41 - 50	*3*	*3*	*6*	*6*	*9*	*9*
51 - 60	*3*	*3*	*6*	*6*	*9*	*9*

Figure 6.1

directory consists of 36 elements (i.e., fragments) and each element is assigned to a processor. For example, the fragment that contains tuples with *ticker_symbol* attribute values that range from letters A through D and *price* attribute values that range from values 21 to 30 is assigned to processor 2.

Next, contrast the execution paradigm of queries A and B when the STOCK relation is hash partitioned on the *ticker_symbol* attribute and when it is declustered using the MDPS (using the assignment presented in Figure 6.1). Query type A is an exact match query on the *ticker_symbol* attribute. The hash partitioning strategy can localize the execution of this query to a single processor. The MDPS directs the execution of this query to three processors because its selection predicate maps to one column of the two dimensional directory and each column is assigned to three different processors. For example, consider a query that retrieves the record that corresponds to American Express (STOCK.ticker_symbol = AXP). The predicate of this query maps to the first column of the grid directory and processors 1, 2, and 3 are used to execute it.

The second query is a range query on the *price* attribute. The hash partitioning strategy must direct this query to all nine processors because: 1) *price* is not the partitioning attribute of the STOCK relation, and 2) even if it was, the hash partitioning strategy directs the execution of range queries to all processors containing fragments of the referenced relation. The MDPS directs this query to three processors since its predicate value maps to one row of the grid directory and each row is assigned to three different processors.

Consequently, the MDPS directs the queries to an average of three processors while the range and hash partitioning strategies would each use an average of five processors. Recall that because of the minimal resource requirements of both queries, a single processor should have been employed for each. Thus, the MDPS reduces the average response time of the queries and increases the overall throughput of the system since it comes closer to using the optimal number of processors to execute both queries.

In this example, the range and hash partitioning strategies both used an average of $\frac{P+1}{2}$ processors to execute the queries while the MDPS used an average of $\sqrt{P}$ (i.e., $\frac{\sqrt{P}+\sqrt{P}}{2}$) processors, where P is the number of processors in the system. Thus, the fraction of the P processors that are freed by the MDPS to perform useful work is $\frac{P+1-2\sqrt{P}}{2P}$ (e.g., in a nine processor configuration, the MDPS frees up 22.2% of the total processing capability of the system that was required by the range and hash partitioning strategies to execute the mix of two queries). If the number of processors in the multiprocessor and the size of the STOCK relation (i.e., number of fragments of the relation in the grid directory) are increased proportionally, the fraction of processors that are freed by the MDPS also increases (see Figure 6.2). Thus, for a system with many processors, the MDPS will substantially improve the overall throughput of the system. □

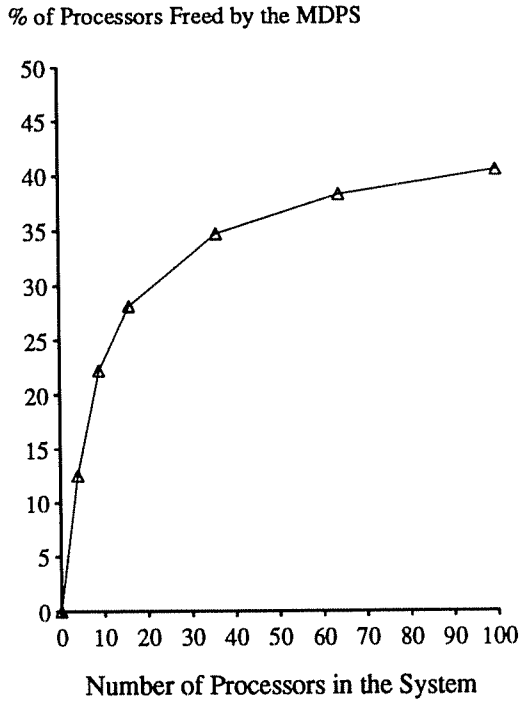


Figure 6.2: Percentage of Processors Freed by the MDPS

Note that in this example the MDPS directs query type A to three processors in order to make it possible to localize the execution of query type B. Thus, it outperforms the range and hash declustering strategies when the whole workload is considered and does not necessarily minimize the response time of each query individually. For example, when compared with the hash declustering strategy, the MDPS actually will have a higher response time and a lower throughput for query type A because it uses three processors while the hash partitioning strategy uses a single processor.

In the next section, we provide a brief overview of the MDPS algorithm for constructing a multidimensional directory on a relation. In the subsequent sections, we expand on this overview and explain the details of the algorithm.

6.1. Overview of the MDPS Algorithm

Given a relation to be declustered using the MDPS, we assume that the database administrator provides the resource requirements and frequency of occurrence of each query accessing this relation, and specifies the K partitioning attributes. Based on this information and the processing capability of the system, the MDPS computes the number of processors (termed M) that should be used to execute an average query (termed Q_{Ave}) representative of the different queries in the workload. In order to ensure that M processors are used to execute the Q_{Ave}, the MDPS computes the cardinality of each fragment of the relation to be $\frac{1}{M}$ number of tuples processed by the average query. Thus, the selection predicate of Q_{Ave} will cover M fragments. By assigning each of the M fragments to a different processor, the MDPS will use the correct number of processors to execute Q_{Ave}. In addition to M, the MDPS computes M_i which represents the number of processors that should be used to execute those queries whose predicate includes attribute i. It is computed based on the resource requirements of this subset of queries and is used to determine the splitting strategy for constructing a grid directory on the relation.

Next, the MDPS inputs to the grid file algorithm [NIEV84] the following information: 1) the cardinality of each fragment, 2) the splitting strategy, 3) the relation to be declustered, and 4) the K partitioning attributes. The grid file algorithm scans the relation and constructs a K dimensional grid directory whose ith dimension consists of N_i ranges of partitioning attribute values. We term each range of a partitioning attribute value a *slice* (e.g., the *ticker_symbol* attribute in Figure 6.1 consists of six slices).

The MDPS then analyzes the grid directory and assigns its elements (i.e, fragments of the relation) to the processors in the system. The assignment of elements to processors is a complex task because the algorithm attempts to satisfy two conflicting goals simultaneously. On one hand, M_i different processors should be assigned to each slice in dimension i so that the optimizer will use M_i processors to execute those queries that access attribute i. On the other hand, the elements of the grid directory should be distributed evenly among the processors in order to ensure that the full processing capability of the system is used while executing queries accessing this relation[1]. These two goals conflict because the first advocates assigning the fragments to a subset of processors (M_i) while the second goal advocates distributing the fragments among all processors. In Section 6.5, we present our heuristic solution for assigning the elements to the processors.

During the last stage, the MDPS scans the relation a second time and assigns tuples to processors based on the contents of the grid directory. The grid directory is then stored in the database catalog and used by the query optimizer to localize the execution of those queries whose selection predicate is applied to one or more of the partitioning attribute.

The rest of this chapter is organized as follows. Section 6.2 describes how we compute the values of M, M_i, and the cardinality of a fragment. In Section 6.3, we present the splitting strategy to construct a grid directory. Section 6.4 repeats from [NIEV84] the steps involved to

---

[1] Assuming a uniform distribution of access to the tuples of a relation, the load is balanced by distributing elements evenly across processors.

create a grid directory on a relation (i.e., the grid file algorithm). Section 6.5 describes a heuristic for assigning the elements of the directory to the processors. In Section 6.6, we present the other advantages of the MDPS.

6.2. Cardinality of a Fragment

The MDPS determines the cardinality of each fragment of a relation based on the processing capability of the system and the resource requirements of the queries that access a relation. By using this information, the MDPS provides the appropriate degree of intra-operator parallelism for the selection operators that access the relation.

In order to simplify the discussion, we first define how the cardinality of a fragment is determined when the MDPS uses a single attribute to decluster a relation. At the end of this section, we then consider the impact of using multiple partitioning attributes on the cardinality of a fragment. We discuss the details in the context of a mixed workload consisting of n selection queries. For each query Q_i, the workload defines the CPU processing time (CPU_i), the Disk processing Time ($Disk_i$), and the Network Processing time (Net_i) of that query. Observe that these times are determined based on the resource requirements of each query and the processing capability of the system. Each query retrieves and processes $TuplesPerQ_i$ tuples from the database. Furthermore, we assume that the workload defines the frequency of occurrence of each query ($FreqQ_i$).

Rather than deriving the cardinality of a fragment with respect to each query in the workload, we define an average query (Q_{Ave}) that represents the queries that access the partitioning attributes. The CPU, disk and network processing quanta for this query are:

$$CPU_{Ave} = \sum_{i=0}^{n} (CPU_i * FreqQ_i)$$

$$Disk_{Ave} = \sum_{i=0}^{n} (Disk_i * FreqQ_i)$$

$$Net_{Ave} = \sum_{i=0}^{n} (Net_i * FreqQ_i)$$

$$TuplesPerQ_{Ave} = \sum_{i=0}^{n} (TuplesPerQ_i * FreqQ_i)$$

We will use this query throughout our discussion.

The three principal resources consumed during the execution of Q_{Ave} are: 1) CPU, 2) disk I/O , and 3) communications. The amount of each is dependent on the processing capability of the system and the resource requirements of the query. Assume that a single processor cannot overlap the use of two resources for an individual query[2]. Thus, the execution time of Q_{Ave} on a single processor in a single user environment is:

$$ExecutionTime = CPU_{Ave} + Disk_{Ave} + Net_{Ave} \quad (1)$$

As data is partitioned and additional processors are used to execute the query, the response time of the query decreases proportionally[3]. However, the overhead incurred by using an additional processor (term this variable *CP*) must be taken into account. This overhead is primarily in the form of additional messages to control the execution of the query on additional processors and is a function of the number of processors used.[4] In Gamma, this overhead is a linear function of the number of processors.

---

[2] The justification for this assumption is provided at the end of this section.

[3] The linear speedup results presented in [DEWI90] justify this assumption.

[4] The algorithm used to schedule and commit a multi-processor query defines this function. For example, if a tree activation protocol is used, this function will be logarithmic in the number of processors participating in the execution of the query (where the base of the log is the branching factor of the tree).

The response time (RT) of a query as a function of the number of processors (termed M) used to execute it can be defined as follows:

$$RT(M) = \frac{CPU_{Ave} + Disk_{Ave} + Net_{Ave}}{M} + M * CP \quad (2)$$

The first goal of the MDPS is to decluster a relation across M processors in order to minimize the response time for the query. Setting the first derivative of the function $RT(M)$ to zero, one can solve for the desired value of M:

$$M = \left[\frac{CPU_{Ave} + Disk_{Ave} + Net_{Ave}}{CP} \right]^{\frac{1}{2}} \quad (3)$$

If the relation is declustered across M processors (where M is rounded to the nearest integer), the response time for Q_{Ave} will be minimized.

While partitioning a relation across M processors minimizes the response time for Q_{Ave} in a single user environment, it does not necessarily maximize the throughput of the system in a multiuser environment. For example, assume that $M = 1$ (i.e., that the resource requirements of the queries in the workload are minimal) and that the workload consists of a single query that accesses the STOCK relation using the *price* attribute. Thus, the relation is stored entirely at one processor. Consider also the performance of the system when the relation is range partitioned across P processors (say $P = 5$) using the *price* attribute. The range partitioning strategy, by its nature, will direct the queries in the workload to a single processor most of the time. Consequently, the average response time of the query in a single user environment is almost the same for both partitioning strategies. However, in a multiuser environment, the range partitioning strategy will distribute the concurrently executing queries among all five processors, while the MDPS will always direct them to a single processor. Thus, the throughput of the system will almost certainly be higher with the range partitioning strategy.

The throughput of the system can be maximized by changing the interpretation of M. Instead of M representing the number of processors over which a relation should be declustered, M should represent the number of processors that should participate in the exe-

cution of Q_{Ave}. Since Q_{Ave} processes $TuplesPerQ_{Ave}$ tuples, each fragment of the relation should contain $FC = \frac{TuplesPerQ_{Ave}}{M}$ tuples. In the case of a one-dimensional MDPS, by insuring that M adjacent fragments are assigned to different processors, the MDPS can ensure that at least M and at most $M + 1$ processors are employed to execute the queries in the workload[5].

In order to demonstrate that this is indeed the desired criteria for creating the fragments of a relation, consider the following example. Assume a 10,000 tuple relation with unique values for the partitioning attribute ranging from 0 to 9,999 and an "average" query which accesses this relation using a range predicate on a single attribute to retrieve and process 500 tuples ($TuplesPerQ = 500$). Also, assume that the optimal performance is achieved when 5 processors are used ($M = 5$) to execute this query. Thus, the cardinality of each fragment is 100 tuples ($FC = \frac{TuplesPerQ}{M} = 100$). The MDPS declusters the relation on a single attribute and constructs a one-dimensional grid directory that is identical to a range table created by the range partitioning strategy.

Thus, the MDPS will partition the relation into 100 fragments and creates a grid directory that consists of 100 entries (see Figure 6.3). By assigning the fragments to the processors in a round-robin fashion, we can insure that M adjacent fragments are assigned to different processors. Below, we compare and contrast this declustering strategy with the range and hash

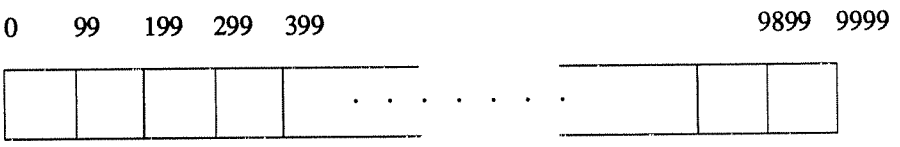


Figure 6.3

---

[5] The MDPS utilizes $M + 1$ processors more often than M processors. This limitation can be solved by adjusting the cardinality of a fragment (FC) to be $FC = \frac{TuplesPerQ_{Ave}}{M - 1}$. The impact of this limitation and its possible solutions remains as a part of the future work of this thesis to be studied along with a sensitivity analysis of M.

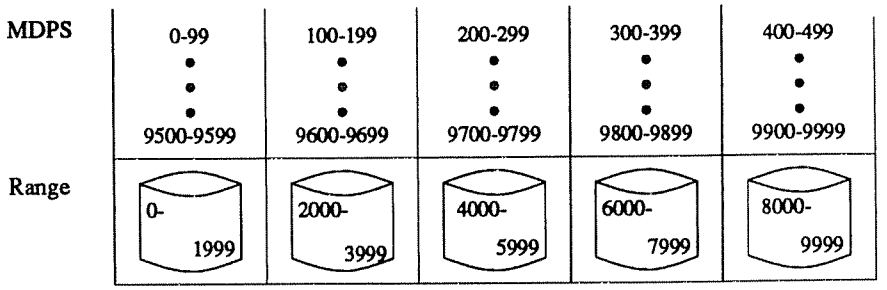
partitioning strategies.

First, consider the case where *P*, the number of processors, is equal to 5 (Figure 6.4.a). With the MDPS, the range of a query will overlap either 5 or 6 fragments and in each case is directed to all *P* processors. The hash partitioning strategy will also use all *P* processors since it cannot localize the execution of range queries. The range partitioning strategy will decluster the relation into 5 fragments. Since the range of a query falls within a range of a single fragment most of the time and overlaps the range of two fragments some of the time, the query will be directed to either 1 or 2 processors.

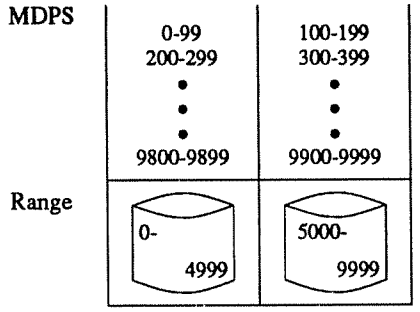
When *P*, the number of processors, is less than *M* (see Figure 6.4.b), the MDPS will still use all *P* processors to execute a query because it enforces the constraint that the *M* adjacent fragments be assigned to different processors whenever possible. Conversely, the range partitioning strategy declusters the relation into 2 (i.e., *P*) fragments, significantly increasing the probability of a query being directed to only one processor.

In the final case, $P > M$ (Figure 6.4.c), the MDPS will distribute the 100 fragments of the relation across all *P* processors to insure that all available resources are used in order to maximize the throughput of the system when executing multiple queries concurrently. However, since the range of a query will overlap only 5 or 6 fragments, each query is directed to almost the optimal number of processors. Conversely, the hash partitioning strategy will direct the query to all *P* processors, incurring the startup, communication, and termination overheads associated with executing the query on more processors than absolutely necessary. The range partitioning strategy will execute the query on only 1 or 2 processors, again using less than the optimal number of processors.

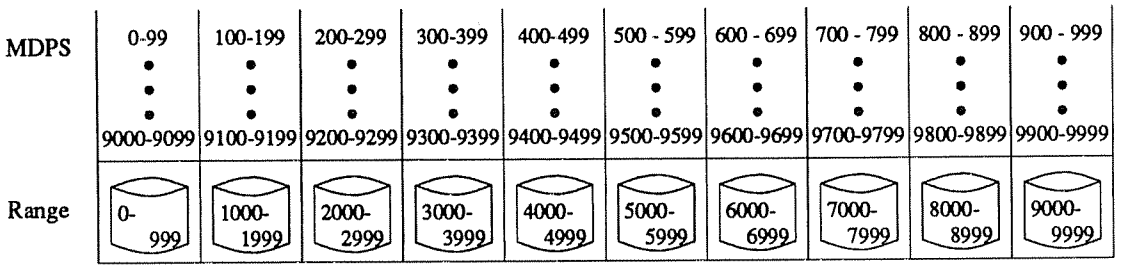
Thus, by utilizing the characteristics of the queries to create the fragments of a relation, the MDPS obtains the appropriate degree of parallelism for those queries. In the worst case, the cardinality of each fragment will equal one (i.e., the number of entries in the directory will equal the cardinality of the relation). This might be acceptable if the resource requirements of the query are extremely high (e.g., image processing). However, if they are low, the overhead of



a. M = P



b. M > P



c. M < P

Figure 6.4

searching the one dimensional directory might become a significant fraction of the query's execution time. Thus, this overhead must be considered when determining the size of each fragment. The number of entries in the one dimensional grid directory is $\frac{M * Cardinality(Rel)}{TuplesPerQ}$. If a lookup in this directory is performed using a linear search, the response time of the query is defined by the following equation:

$$RT(M) = \frac{CPU+Disk+Net}{M} + M * CP + \frac{M * Cardinality(Rel) * CS}{TuplesPerQ}$$

where CS represents the overhead associated with searching a single entry in the directory. Hence, the number of processors used to execute the query is specified by the following formula:

$$M = \left\lceil \frac{CPU + Disk + Net}{CP + \frac{Cardinality(Rel) * CS}{TuplesPerQ}} \right\rceil^{\frac{1}{2}} \quad (4)$$

If a binary search algorithm is used instead to process accesses to the grid directory, M becomes:

$$M = \frac{-\frac{CS}{\ln 2} + \left\lceil (\frac{CS}{\ln 2})^2 + (4 * CP * (CPU+Disk+Net)) \right\rceil^{\frac{1}{2}}}{2 * CP}$$

One simplifying assumption that we have made is that a query does not use multiple resources on a single processor simultaneously. This however is not a realistic assumption. For example, in Chapters 4 and 5, we repeatedly observed the impact of the read-ahead process that overlaps disk and CPU operations. Our justification for this simplification is that in a multiuser environment, the probability of overlap within a single query is fairly low since the total number of resources is relatively small compared to the multiprogramming level. Thus, while one query is consuming one of the resources (e.g. the CPU), it is highly likely that other queries will be consuming the other resources.

Thus far, we have described how the MDPS declusters a relation with respect to the resource requirements of Q_{Ave}. We now expand our discussion and consider the n individual queries that constitute the workload for a relation. In this case, the response time $RT(M)$ becomes:

$$RT(M) = \sum_{i=0}^{n} (\frac{CPU_i+Disk_i+Net_i}{M} + M * CP) * FreqQ_i \quad (5)$$

Thus,

$$M = \left\lceil \frac{\sum_{i=0}^{n} (CPU_i + Disk_i + Net_i) * FreqQ_i}{CP} \right\rceil^{\frac{1}{2}}$$

and the *FC* for each fragment is:

$$\frac{\sum_{i=0}^{n} (TuplesPerQ_i * FreqQ_i)}{M} \tag{6}$$

One assumption of this discussion has been that the MDPS computes the cardinality of each fragment using the resource requirements of all queries accessing a relation. However, it appears more appropriate to create the fragments based on the resource requirements of only those queries that access the partitioning attribute (termed PA_Queries) since those queries that do not access the partitioning attribute (termed NPA_Queries) are directed to all processors regardless of how the fragments are created. Ignoring the NPA_Queries is appropriate only if the resulting number of fragments is greater than the number of processors in the system (it is appropriate only if $\frac{Cardinality\ (Rel)}{FC_{PA\_Queries}} \geq P$). To illustrate this, assume a system composed of 10 processors. While the resource requirement of PA_Queries dictate the utilization of a single processor ($M = 1$), the resource requirements of NPA_Queries requires the participation of 10 processors ($M = 10$). In addition, assume that the workload is composed of 99% type NPA_Queries and 1% type PA_Queries. If by utilizing the resource requirements of PA_Queries, the MDPS creates less than 10 fragments, then the overall throughput of the system is significantly reduced because 99% of the queries are directed to fewer processors than what their resource requirements dictated. If, however, 10 or more fragments are formed when the relation is declustered, then ignoring the NPA_Queries might actually enhance the overall throughput of the system because: 1) the throughput of the NPA_Queries is not affected since they are directed to ten processors regardless of the range of each partitioning attribute assigned to each processor and the number of entries in the grid directory, and 2) the throughput of the system for PA_Queries might increase because the number of tuples per relation fragment is determined entirely by these queries, increasing the probability of

PA_Queries being directed to a single processor.

When the MDPS declusters a relation using multiple attributes, the Q_{Ave} will consist of those queries which access the relation via a selection predicate on the K partitioning attributes (or all queries, depending whether or not the number of fragments created based on the PA_Queries is greater than the number of processors). Based on the resource requirements of these queries and the processing capability of the system, the value of M is determined. Using this value, we determine the cardinality of each fragment.

In addition to calculating the value of M, we also compute the value of M_i for each partitioning attribute i (i.e., each dimension of the grid directory). This value is computed as follows. Assuming s represents the number of different queries whose selection predicate is applied to attribute i, we first compute the frequency of occurrence of each query relative to the total frequency of queries accessing this attribute (termed RelFreqQ). For each query j, the relative frequency of that query is:

$$RelFreqQ_j = \frac{FreqQ_j}{\sum_{r=1}^{s} FreqQ_r}$$

The value of M_i for attribute i is:

$$M_i = \left\lceil \frac{\sum_{r=1}^{s} (CPU_r + Disk_r + Net_r) * RelFreqQ_r}{CP} \right\rceil^{\frac{1}{2}}$$

The value of M_i represents the number of different processors that should appear in each slice of dimension i (e.g., in each column of the grid in Figure 6.1). We use the value of M_i in both Section 6.3 to determine the splitting strategy for constructing the grid directory and also in Section 6.5 to assign the elements of the grid directory to processors.

6.3. Strategy Used to Split the Alternative Dimensions

In order to construct a grid directory on a relation, the grid file algorithm requires, as one of its inputs, the ratio of splits along each dimension of the grid file (this will be clarified in the next section). Intuitively, the splitting strategy determines the shape of the grid directory, in

particular, the value of N_i for $i = 1$ to K.

In order to demonstrate this, recall the STOCK relation, the queries accessing this relation, and the grid directory on this relation (see Figure 6.1). Instead of a system with 9 processors, assume a 36 processor configuration. Since the frequency of access and the value of M_i for each partitioning attribute is identical, the MDPS constructs a square directory (see Figure 6.5) and assigns each element to a different processor. The range and hash declustering strategies also decluster the relation across all 36 processors. Assume that these declustering strategies use the *ticker_symbol* as the partitioning attribute. Consequently, they can localize the execution of query type A to a single processor, while employing all 36 processors to execute query type B. Thus, each will use an average of 18.5 (i.e., $\frac{36+1}{2}$) processors to execute the queries in the workload. The MDPS employs six processors to execute each query since the

Ticker_Symbol

PRICE	A-D	E-H	I-L	M-P	Q-T	U-Z
0 - 10	1	2	3	4	5	6
11 - 20	7	8	9	10	11	12
21 - 30	13	14	15	16	17	18
31 - 40	19	20	21	22	23	24
41 - 50	25	26	27	28	29	30
51 - 60	31	32	33	34	35	36

Figure 6.5

predicate value of each query maps to either a row or a column of the grid directory and each row or column is assigned to 6 different processors. Thus, the throughput of the system with the MDPS will be higher than with either the range or hash declustering strategies.

However, if the workload is composed of 90% type A queries and 10% type B queries, then a square directory is no longer appropriate. For example, with the directory in Figure 6.5, the range and hash partitioning strategies will outperform the MDPS because they use an average of 4.5 processors (90% * 1 + 10% * 36) while the MDPS continues to use an average of 6 processors (90% * 6 + 10% * 6). By constructing a directory such that the number of slices for each dimension (i.e., attribute) is proportional to the frequency of access to that dimension, this scenario will not occur. For example, the dimension corresponding to the *ticker_symbol* attribute should have nine times as many slices as the *price* attribute due to the significant difference in the frequency of access to each attribute domain. By requiring the grid file algorithm to split the different dimensions proportional to their frequency of access, the grid directory shown in Figure 6.6 will be produced. With this directory, the MDPS employs an average of 3.6 processors (90% * 2 + 10% * 18) and results in a higher throughput than either the range or hash partitioning strategy.

When the values of M_i are not the same for all K dimensions, their values must be taken into consideration in order to ensure that each slice of dimension i has at least M_i elements, so that M_i different processors can be assigned to each slice of dimension i. To achieve this, the

Ticker_Symbol

		A–A	B–B	C–C	D–D	E–E	F–F	G–G	H–H	I–I	J–K	L–M	N–O	P–Q	R–R	S–T	U–V	W–X	Y–Z
Price	*0 - 30*	1	3	5	7	9	11	13	15	17	19	21	23	25	27	29	31	33	35
	31 - 60	2	4	6	8	10	12	14	16	18	20	22	24	26	28	30	32	34	36

Figure 6.6

number of slices for each dimension should be proportional to the *Percentage_Splits* defined by equation (7):

$$Percentage\_Splits = FreqD_i * \left(\frac{\sum_{j=1}^{K} M_j - M_i}{\sum_{j=1}^{K} M_j}\right) \quad (7)$$

This results in a directory with at least M_i elements for each slice in dimension D_i. For example, assume $M_{ticker\_symbol} = 3$ and $M_{price} = 1$. Also, assume that 90% of queries access the *ticker_symbol* attribute and 10% access the *price* attribute. In this case, using Equation 7, the percentage of splits along the *ticker-symbol* attribute domain will be 22.5% and that of the *price* attribute will be 7.5%. Consequently, the *ticker-symbol* attribute should have 3 times as many slices as the *price* attribute, resulting in the directory shown in Figure 6.7[6]. If the value of M_i was not taken into consideration, the resulting grid would have been identical to that of Figure 6.6, causing 2 processors to be employed (instead of 3) to execute the queries with predicates applied to the *ticker-symbol* attribute.

Ticker_Symbol

Price												
	1	*4*	*7*	*10*	*13*	*16*	*19*	*22*	*25*	*28*	*31*	*34*
	2	*5*	*8*	*11*	*14*	*17*	*20*	*23*	*26*	*29*	*32*	*35*
	3	*6*	*9*	*12*	*15*	*18*	*21*	*24*	*27*	*30*	*33*	*36*

Figure 6.7

[6] In this figure, the number of slices along the *ticker-symbol* is actually four times that of the *price* attribute. With this ratio of *percentage_splits*, the grid file algorithm will construct either a 3x12 or a 4x9 directory. In either case, the number of elements for each slice of the *ticker-symbol* attribute will be greater than or equal to $M_{ticker-symbol}$.

6.4. Create the Grid File Directory

Using the cardinality of each fragment from Equation 4 and the frequency of splits (i.e., the desired ratio of slices for the different dimensions) from Equation 7 as input, the grid file algorithm [NIEV84] scans the relation and constructs a grid directory on the relation. Below, we briefly describe the grid file algorithm, repeating from [NIEV84] the steps involved in creating one.

A grid file is a dynamic file structure designed to support multikey access to records. It provides effective support for range queries and queries with partially-specified predicates. This file structure manages a disk with fixed size storage units, usually termed "disk blocks". We use the terminology provided by [NIEV84] and term each storage unit a **bucket.** We assume an unlimited number of buckets, each with a capacity of c records. The grid directory maintains the dynamic correspondence between grid blocks in the record space and data buckets.

The creation of the grid file is best explained with an example, and in order to simplify the description, we present the two dimensional case only. Instead of showing the grid directory which has a one-to-one correspondence with the record space, we draw the bucket pointers from the record space to the appropriate bucket. For example, in Figure 6.8 the *ticker-symbol* and *price* attributes form the axis of a two dimensional record space which points to bucket A (each R represents a record).

The capacity of each bucket is defined by the cardinality of each fragment (using Equation 4). Assume that the capacity of each bucket is three records. Initially, the relation is scanned and three records are assigned to bucket A (see Figure 6.9). When an additional record is inserted (R_4), bucket A overflows and the record space is split along the ticker-symbol attribute domain, a new bucket B is allocated and those records that lie in one half of the record space are moved from the old bucket to the new one (see Figure 6.10). If bucket A overflows again, the record space is split according to a splitting policy. Assume the simplest splitting policy, that of alternating between the dimensions of the record space. Thus, the record space is split along the *price* attribute domain and the records are moved into a new bucket C (see Figure

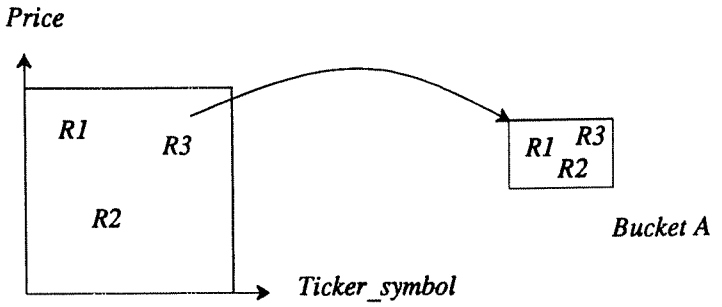


Figure 6.8

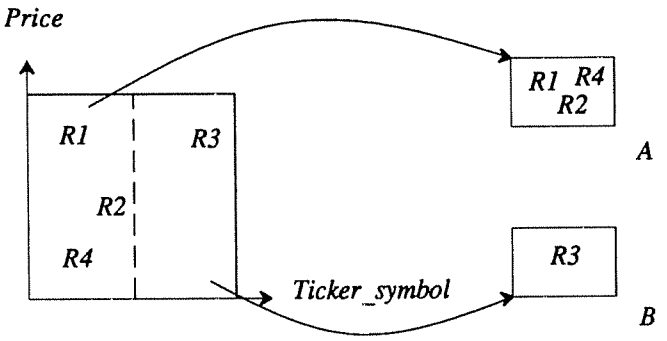


Figure 6.9

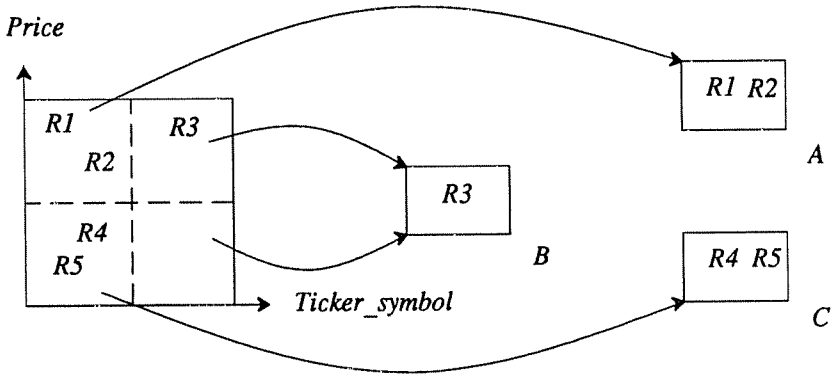


Figure 6.10

6.10). Note that, since bucket B did not overflow, it is left alone. It now corresponds to two regions of the record space (or two elements in the grid directory). As more tuples are inserted into the file, more buckets are allocated and the record space is split appropriately. Thus, the

grid file algorithm creates a 2x2 directory. In the context of the MDPS, each element of the directory corresponds to a fragment of the relation.

A cyclic splitting strategy results in $N_i = N_j$, for $i, j = 1$ to N (i.e., the same number of slices for each dimension of the directory) and is appropriate only when the frequency of access and the value of M_i is identical for each dimension. A more appropriate splitting strategy is when each dimension is split proportionally to the value of *Percentage_Splits* (defined by Equation 7) of the different dimensions.

The output of the grid file algorithm is a K dimensional directory whose ith dimension consists of N_i ranges of partitioning attribute values.

6.5. Analyze the Grid Directory and Assign Entries to Processors

In this step, the MDPS analyzes the K dimensional grid directory produced by the previous step and assigns its elements to the processors in the system. This is a complex problem because we are trying to satisfy two conflicting goals simultaneously. On one hand, M_i different processors should be assigned to each slice of dimension D_i in order to obtain the appropriate degree of parallelism for those queries whose selection predicate is applied to the attribute corresponding to that dimension. On the other hand, the elements of the grid directory should be distributed evenly among the processors in order to use the full processing capability of the system. Assuming a uniform distribution of access to the tuples of a relation, the load is balanced by distributing the elements evenly across multiple processors.

These two goals conflict because the first advocates assigning the fragments to a subset of processors (M_i), while the second goal advocates distributing the fragments among all processors. In some cases it is impossible to satisfy both of these requirements and the DBA must bias the MDPS toward one of these conflicting goals. Since this decision is primarily application dependent, we bias the MDPS towards uniform distribution of tuples across the processors. Using this, we can compare the MDPS with the hash and range declustering strategies.

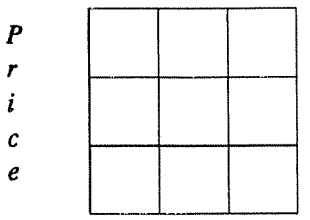
In order to simplify the discussion, we assume that each bucket of the grid directory is 100% occupied. Thus, by assigning the same number of elements to each processor, the MDPS balances the load among the processors (assuming a uniform distribution of access to the tuples). There are two possible scenarios: 1) the number of fragments $|F|$ (i.e., elements in the grid directory) is less than or equal to the number of processors (P) ($|F| \leq P$), or 2) the number of fragments is greater than the number of processors ($|F| > P$). We consider each case individually:

Case 1: $|F| \leq P$

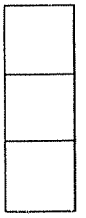
Since the MDPS declusters a relation by using the characteristics of the queries that access the relation, it might create fewer fragments than there are processors in the system (termed partial declustering [COPE88]). In this case, we assign each fragment to a different processor in order to balance the load as evenly as possible. Refer to Figures 6.5, 6.6, and 6.7 and the discussion in Section 6.3 for a comparison of the MDPS to the range and hash declustering strategies when $|F| = P$.

Case 2: $|F| > P$

When the number of elements in the grid directory is greater than the number of processors, the assignment of processors to elements is a complex task. Before describing our solution, we first formalize our terminology. We must assign the elements of a K dimensional directory with N_i ranges of attribute values associated with each dimension D_i. When we fixate on a single range of dimension D_i, we obtain a *slice* of that dimension. There are N_i slices in D_i, each of $K - 1$ dimensions, and each element of the directory belongs to K distinct slices. For example, in the three dimensional directory shown in Figure 6.11.b, the *ticker_symbol* dimension consists of four slices, the P/E dimension has two slices, and the price dimension has three slices. In addition, each element of this directory belongs to three different slices, each from a different dimension. Figure 6.11a shows a two dimensional directory and a slice of the ticker_symbol dimension. Given a query with a predicate on the ticker_symbol attribute, the



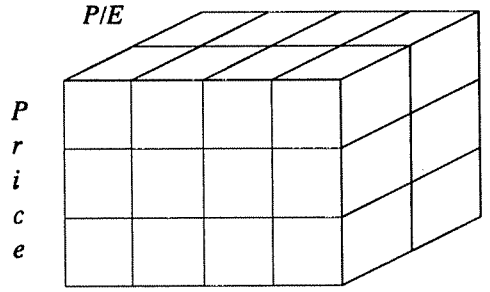
A 2 Dimensional Directory



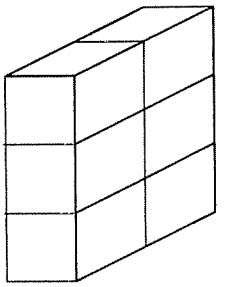
A Slice of the

Dimension Ticker_symbol

a. Two Dimensional Directory



A 3 Dimensional Directory



A Slice of the

Dimension Ticker_symbol

b. Three Dimensional Directory

Figure 6.11

value of the predicate maps to one or more slices of the ticker_symbol dimension. The processors that appear in those slices are used to execute the query.

Using this terminology, the problem can be stated as follows. Given the value of M_i for dimension D_i, and a system with P processors, the elements of the grid directory should be assigned to the processors such that the following two constraints are satisfied: 1) M_i unique processors appear in each of the N_i slices of dimension D_i (in order to ensure that the

appropriate number of processors are used to execute queries in that dimension), and 2) each processor is assigned $\frac{\prod_{i=1}^{K} N_i}{P}$ elements[7] in order to distribute the elements (i.e., fragments) of the grid directory evenly across the processors. For example, the grid directory in Figure 6.1 is two dimensional (K = 2), with 6 slices per dimension ($N_1 = N_2 = 6$). In a 9 processor system, 4 elements (i.e., fragments) are assigned to each processor ($\frac{6 * 6}{9}$). Three different processors appear in each slice of the directory. Thus, three processors are used to execute a query with an exact-match predicate accessing either dimension.

If $K = 1$ (i.e., the relation is declustered on a single attribute), assigning elements to processors in a round-robin fashion satisfies both of these constraints (see Section 6.2 for an example). In Chapter 7, we will compare the performance of the one-dimensional version of MDPS with the range and hash declustering strategy and demonstrate its superiority.

When $K > 1$, the problem can be posed as an integer programming (IP) problem which can be solved for an optimal solution ([GHAN90c]). However, an exhaustive search of the solution space is performed and, for even small grids (e.g., a 5x5 grid), the time required is unacceptable. In [GHAN90c], we present the proof for the theoretical lower bound on the minimum number of processors that can be employed by the MDPS and describe an algorithm that can achieve this theoretical lower bound when: 1) the directory is two dimensional ($K = 2$), 2) N_1 and N_2 are multiples of P, and 3) each dimension has the same frequency of access. When these constraints are removed, we are not aware of an algorithm that can achieve the theoretical lower bound in a reasonable amount of time. Thus, we propose a heuristic that can approximate the optimal solution. We evaluate this heuristic by comparing the quality of its

---

[7] $\prod_{i=1}^{K} N_i$ represents the product of slices and translates into $N_1 * N_2 * ..... * N_k$.

assignment to the theoretical lower bound[8].

We present our heuristic in three steps. In Step 1, we describe an algorithm that obtains an optimal assignment of elements when three constraints are satisfied. In Steps 2, 3, and 4, we successively remove the constraints and present the heuristic for performing the assignment.

STEP 1:

Assume that the following three constraints are satisfied: 1) $\prod_{i=1}^{K} M_i = P$, 2) N_i is a multiple of M_i, and 3) the value of M_i is the same for each dimension D_i (i.e., $M_i = P^{\frac{1}{K}}$, using the first constraint). By partitioning each dimension of the original grid into M_i groups of $\frac{N_i}{M_i}$ slices, the original grid is partitioned into P coarser K-dimensional grids. Intuitively, this is true because the algorithm partitions each dimension M_i times and $\prod_{i=1}^{K} M_i$ determines the total number of grids formed; this is equivalent to P (using the first constraint).

Lemma:

By assigning each coarser grid to a different processor, the MDPS obtains an optimal solution.

Proof:

The original grid can be partitioned into P coarser grids because $\frac{\prod_{i=1}^{K} N_i}{\prod_{i=1}^{K} \frac{N_i}{M_i}} = \prod_{i=1}^{K} M_i = P$ (using constraints one and two). By assigning each grid to a unique processor, the elements will be evenly distributed across the processors because each processor is assigned

[8] As we will describe in Section 6.5.1, the theoretical lower bound used for this evaluation is not a tight bound - i.e., for any arbitrary grid directory, a solution does not necessarily exist for this lower bound. At this point in time, we are not aware of a theoretical lower bound that can ensure the existence of a solution.

$\prod_{t=1}^{K} \frac{N_t}{M_t} = \frac{\prod_{t=1}^{K} N_t}{P}$ elements (using the first constraint). At the same time, each slice of dimension D_t will be assigned $\frac{\prod_{j=1}^{K} M_j}{M_t} = \frac{P}{P^{\frac{1}{K}}} = P^{\frac{K-1}{K}}$ unique processors (using constraints one and three).

When $K = 2$, the proof is straightforward because each slice is assigned $P^{\frac{1}{K}} = M_t$ unique processors (using constraint three) which is the ideal number of processors. When $K > 2$, [GHAN90c] contains the proof for $P^{\frac{K-1}{K}}$ as the theoretical lower bound on the number of unique processors that can be assigned to each dimension. □

Below, we provide an example to illustrate this assignment of elements to processors.

Example 2:

Recall the grid directory from Figure 6.1 ($K = 2$, $N_1 = N_2 = 6$). Assume $M_1 = M_2 = 3$ (instead of 1 in the initial example in order to satisfy the first constraint for illustration purposes). Furthermore, assume a system consisting of 9 processors ($P = 9$). Consequently, $\prod_{t=1}^{2} M_t = P$. In addition, N_t is an even multiple of M_t ($\frac{N_1}{M_1} = \frac{N_2}{M_2} = 2$). Thus, the original 6x6 grid is partitioned into nine 2x2 grids. Each coarser grid is assigned to a unique processor, resulting in an even distribution of elements across the processors and the appropriate number of unique processors in each slice (see Figure 6.1 for the final assignment). □

STEP 2:

We now remove the third constraint and consider the case where the value of M_t is not required to be the same for each dimension D_t. In this case, each M_t is replaced by $\frac{\prod_{j=1}^{K} M_j}{M_t}$. Since each dimension is partitioned into M_t groups of slices, which results in $\frac{\prod_{j=1}^{K} M_j}{M_t}$ unique processors appearing in each dimension, by replacing the value of M_t with $\frac{\prod_{j=1}^{K} M_j}{M_t}$, we ensure

that $\frac{\prod_{j=1}^{K} M_j}{\frac{\prod_{j=1}^{K} M_j}{M_t}} = M_t$ unique processors appear in each slice of D_t.

Example 3:

Assume a 12x8 grid directory and a system composed of 8 processors ($N_1 = 12$, $N_2 = 8$, $P = 8$). Furthermore, assume $M_1 = 4$ and $M_2 = 2$. Since the values of M_t are different, the heuristic replaces the value of M_1 with 2 and M_2 with 4. Using Step 1, the original grid is partitioned into 8 coarser 6x2 grids. By assigning each of these grids to a unique processor, the heuristic obtains an optimal solution (see Figure 6.12). Observe that if the value of M_t was not modified, the resulting assignment would have resulted in the incorrect number of processors being used to execute the queries accessing the different dimensions (see Figure 6.13).

1	1	3	3	5	5	7	7
1	1	3	3	5	5	7	7
1	1	3	3	5	5	7	7
1	1	3	3	5	5	7	7
1	1	3	3	5	5	7	7
1	1	3	3	5	5	7	7
2	2	4	4	6	6	8	8
2	2	4	4	6	6	8	8
2	2	4	4	6	6	8	8
2	2	4	4	6	6	8	8
2	2	4	4	6	6	8	8
2	2	4	4	6	6	8	8

Figure 6.12

1	1	1	1	5	5	5	5
1	1	1	1	5	5	5	5
1	1	1	1	5	5	5	5
2	2	2	2	6	6	6	6
2	2	2	2	6	6	6	6
2	2	2	2	6	6	6	6
3	3	3	3	7	7	7	7
3	3	3	3	7	7	7	7
3	3	3	3	7	7	7	7
4	4	4	4	8	8	8	8
4	4	4	4	8	8	8	8
4	4	4	4	8	8	8	8

Figure 6.13

STEP 3:

We now relax the second constraint and consider the case where N_i is not an even multiple of M_i. In this case, a divide and conquer approach is used to assign processors to the elements of the grid directory. Figure 6.14 presents the four phases of the heuristic. In order to clarify each phase, we present an example and show the assignment of the elements of its directory after each phase of the heuristic. For this example, recall the parameters from Example 2 ($P = 9$, $M_1 = M_2 = 3$). However, this time assume that the grid file algorithm produces an 11x7 directory ($N_1 = 11$, $N_2 = 7$).

Phase 1

/* for each dimension i, compute $NewN_i$ to be an even multiple of M_i */
for $i = 1$ to K
{
    NewN[i] = N[i] - (N[i] MOD M[i])
}
Using Step 1, assign the coarser K-dimensional grid with NewN[i] slices

Phase 2

$$Remaining\_Elmts = \prod_{i=1}^{K} N[i] - \prod_{i=1}^{K} NewN[i]$$

for $i =$ 1 to P

$$QuotaOf\_P[i] = \left\lfloor \frac{Remaining\_Elmts}{P} \right\rfloor$$

for $i =$ the most frequently accessed dimension to the least frequently accessed dimension
{
    $Elmts =$ Number of unassigned elements in a slice of dimension i
    for $Z =$ each slice in dimension i of the grid with NewN[i] slices
    {
        initialize set A to the unique processors that appear in slice Z
        remove from set A the processors whose quota has been reached
        if ($\sum QuotaOf\_P[a] \geq Elmts$, $a \in A$)
        {
            for $j =$ each unassigned element of slice Z
            {
                For each processor in set A determine if it appears in any of the other K slices
                    that include j as an element
                if a processor in set A appears in all $K - 1$ slices, term it x
                if no single processor in set A appears in all $K - 1$ slices,
                    find a processor in set A which matches a slice corresponding to a dimension with the highest frequency of access, term it x

```
                if no match can be found, find the processor in set A with
                    the highest quota, term it x
                assign processor x to element j
                decrement QuotaOf_P[x]
                decrement Remaining_Elmts
                if (QuotaOf_P[x] = 0)
                    remove x from set A
            }
        }
    }
}
```

if (*Remaining_Elmts* = 0) we are done

Phase 3

$$Remainder = (\prod_{i=1}^{K} N[i] - \prod_{i=1}^{K} NewN[i]) \quad MOD \quad P$$

```
for i = the slice with the least number of unassigned elements
{
    c = the dimension this slice belongs to
    initialize set A to the unique processors that appear in slice i
    remove the processors whose quota equals -1 from set A
    if (Remainder = 0)
        remove from set A the processors whose quota equals 0
    if ((quota of processors in this set) + minimum(remainder,number
    of processors in set A whose quota = 0) ≥ Elmts)
    {
        for j = each unassigned element of slice i
        {
            For each processor in set A determine if it appears in any of
                the other K – 1 slices that include j as an element
            if a processor in set A appears in all K – 1 slices, term it x
            if no single processor in set A appears in all K – 1 slices, find
                a processor in set A which matches a slice corresponding to
                a dimension with the highest frequency of access, term it x
            if no match can be found, find the processor in set A with the
                highest quota, term it x
            assign processor x to element j
            decrement QuotaOf_P[x]
            decrement Remaining_Elmts
            if (QuotaOf_P[x] = –1)
            {
                decrement Remainder
                remove x from set A
            }
            if (Remainder = 0)
                remove all processors from set A whose quota equals zero
        }
    }
}
```

Phase 4

```
Initialize set A to processors whose quota ≠ −1
if (Remainder = 0)
    remove all processors from set A whose quota equals zero
for each unassigned element j
{
        For each processor in set A determine if it appears in any
            of the K slices that include J as an element
        if a processor in set A appears in all K slices, term it x
        if no single processor in set A appears in all K slices, find a
            processor in set A which matches a slice corresponding to a
            dimension with the highest frequency of access, term it x
        if no match can be found, find the processor in set A with the
            highest quota, term it x
        assign processor x to element j
        decrement QuotaOf_P[x]
        if (QuotaOf_P[x] = −1) decrement Remainder
        if (Remainder = 0)
            remove all processors from set A whose quota equals zero
}
```

Figure 6.14: Heuristic Solution for Step 3

During Phase 1, the heuristic reduces the original directory to a smaller one that satisfies the second constraint by reducing N_i to be a multiple of M_i (for i from 1 to K, term this $NewN_i$). The algorithm from Step 1 is then used to assign the elements of this grid whose dimension D_i contains $NewN_i$ slices. In our example, N_1 is reduced to 9 and N_2 to 6. The assignment of the elements of this 9x6 directory after Phase 1 is shown in Figure 6.15.

During Phases 2, 3, and 4, the remaining elements of the directory are assigned. Phases 2 and 3 analyze each slice of the directory and either assign all of its unassigned elements or none, with the primary objective being to obtain M_i different processors for each slice. Both **attempt** to accomplish this by first determining which M_i unique processors already appear in that slice (from the assignment of Phase 1), ensuring that the unassigned elements of this slice are assigned to those processors. This allows the heuristic to obtain M_i unique processors for as many slices as possible before considering elements individually. While Phase 2 assigns the unassigned elements of the first $NewN_i$ slices of each dimension D_i, Phase 3 processes the slices with unassigned elements that remain from Phase 2 in addition to $\sum_{i=1}^{K} (N_i - NewN_i)$ slices

	1	2	3	4	5	6	7
1	1	1	4	4	7	7	
2	1	1	4	4	7	7	
3	1	1	4	4	7	7	
4	2	2	5	5	8	8	
5	2	2	5	5	8	8	
6	2	2	5	5	8	8	
7	3	3	6	6	9	9	
8	3	3	6	6	9	9	
9	3	3	6	6	9	9	
10							
11							

Figure 6.15

that were ignored during Phase 2. For example, Phase 2 only analyzes the unassigned elements of the first 9 rows and 6 columns of the directory in Figure 6.15, while Phase 3 analyzes any rows and columns that remain from Phase 2 in addition to column 7 and rows 10 and 11. Phase 2 does not analyze the $\sum_{i=1}^{K} (N_i - NewN_i)$ slices since none of their elements have been assigned. While perhaps counterintuitive, Phase 2 ends up assigning most of the elements of these slices. Phase 3 then uses these assignments to direct the assignment of the remaining elements of these slices. Phase 4 analyzes each unassigned element and assigns a processor to it. Below, we describe the details of each phase.

During Phase 2, based on the number of unassigned elements in the directory (which is $\prod_{i=1}^{K} N_i - \prod_{i=1}^{K} NewN_i$, termed *Remaining_Elements*), we compute the maximum number of elements that can be assigned to each processor in the system (i.e., the quota of each processor is set to $\left\lfloor \frac{Remaining\_Elements}{P} \right\rfloor$). Starting with the most frequently accessed dimension D_i, the heuristic assigns processors to the unassigned elements of each of the $NewN_i$ slices in that

dimension as follows. First, it determines the unique processors (from Phase 1) that appear in that slice (term this set A). Those processors whose quota have been exhausted are removed from set A. Since the heuristic either assigns all the remaining elements of a slice or none, it checks the cumulative quota of the processors remaining in set A to ensure that it is greater than the number of unassigned elements of the slice. If this check fails, the heuristic continues with the next slice. Otherwise, we take the first unassigned element of the slice and analyze it in the context of the other $K-1$ slices that include it as an element in order to impose further constraints on the qualifying processors in set A. (Recall that each element of the directory is a member of K distinct slices). The motivation for this step is to try to obtain M_i unique processors for as many slices as possible. This can only be achieved by analyzing an unassigned element in the context of all K slices that include it by comparing each processor in set A with the processors that appear in the other $K-1$ slices. If a single processor in set A appears in all $K-1$ slices, the element is assigned to this processor. Otherwise, we assign the element to the processor that matches an element of the slice corresponding to the dimension with the highest frequency of access. If no processor in set A appears in the other $K-1$ slices, the element is assigned to that processor in set A with the highest quota in order to avoid the premature exhaustion of the quota of a processor. Otherwise, the quota of a processor might be reached after processing only a few slices, in effect reducing the number of unique processors in set A when assigning the other slices. By increasing the number of unique processors in set A, the probability of finding a matching processor in the other $K-1$ slices that include this element is also increased.

For our example, Phase 2 has the following impact on Figure 6.15. First, it determines that 23 elements remain to be assigned. To evenly distribute the elements, all but 5 processors should be assigned two elements with the remaining 5 processors being assigned three. Thus, the quota of each processor is set to two. We start with the unassigned elements of the first 9 ($NewN_i$) rows. Processors 1, 4, and 7 qualify for assignment to the unassigned element of the first row. Since all the elements of the column that includes this element are unassigned, and since the quota of each processor is the same, we chose one of the qualifying

processors and assign the element to it; say processor 1. Processors 1, 4, and 7 qualify again for the unassigned element of the second row. However, since processor 1 appears in the column that includes this element, the element is assigned to processor 1, causing the quota of processor 1 to become zero. The same processors qualify again for the unassigned element of the third row. Although processor 1 appears in the column that includes this element, its quota has been reached and no more elements can be assigned to it. Since the quota of the remaining two processors is identical, the heuristic chooses one of them and assigns the element to it; say processor 4. Using this procedure repeatedly, Phase 2 assigns processors to the elements of each of the first 9 rows and 6 columns. The state of the directory at the end of Phase 2 is presented in Figure 6.16. The unassigned elements of columns 1, 2, and 4 remain unassigned because the quota of the processors that qualified for each of those slices was lower than the number of unassigned elements of that slice (recall that Phase 2 either assigns all the elements of a slice or none). The elements of column 7 and rows 10, 11 remain unassigned because Phase 2 does not analyze them. However, note that Phase 2 assigns some of

1	1	4	4	7	7	1
1	1	4	4	7	7	1
1	1	4	4	7	7	4
2	2	5	5	8	8	2
2	2	5	5	8	8	2
2	2	5	5	8	8	5
3	3	6	6	9	9	3
3	3	6	6	9	9	3
3	3	6	6	9	9	6
		4		7	7	
		5		8	8	

Figure 6.16

the unassigned elements of these slices.

After Phase 2, some elements may remain unassigned because: 1) Phase 2 ignores the $N_i - NewN_i$ slices of dimension D_i, 2) Phase 2 either assigns all unassigned elements of a slice or none, and 3) the total number of unassigned elements in the grid may not be an even multiple of the number of processors in the system. During Phase 3, the heuristic sorts the remaining slices based on the number of unassigned elements in each. In addition, if the number of unassigned elements after Phase 1 was not an even multiple of the total number of processors, we compute the remainder (termed R). R defines the number of processors that can be assigned one more element than their fair share (i.e., in the context of the pseudo code presented in Figure 6.14, R defines the number of processors whose quota can become -1). Starting with the slice that has the fewest unassigned elements, the unique processors that appear in that slice are determined (again, term this set A). The processors whose quota are -1 are removed from set A, as no more elements can be assigned to them. If the value of R is zero, all processors with a zero quota are also removed because we have exhausted the processors that can be assigned one more element (fragment) than their fair share. Again, the heuristic either assigns all the remaining elements of a slice or none by comparing the number of unassigned elements of a slice to the sum of the following two variables: 1) the quota of processors in set A, and 2) either R or the number of processors with a zero quota, whichever is lower (this term computes the number of processors that can be assigned more than their fair share of elements while ensuring that there are enough of these processor types in set A). Assuming that this check succeeds, each remaining element of this slice is analyzed in the context of the other $K - 1$ slices that include it and the element is assigned to a processor in set A that appears in the other $K - 1$ slices. If no processor in set A appears in the other $K - 1$ slices, the element is assigned to that processor in set A with the highest quota. Each time an element is assigned to a processor whose quota is zero, we decrement the value of R. As soon as the value of R becomes zero, we remove all processors with a zero quota from set A.

For our example directory, Phase 3 sorts the remaining slices based on their number of unassigned elements. The sorted list includes: columns 1, 2, 4, 7, and rows 10, 11. Columns

1, 2, 4, and 7 are at the head of this list because each has two unassigned elements. The quota of processors 1, 2, 3, 4, 5, 7, and 8 is zero, that of processor 6 is one and processor 9 two. We also determine that five processors (23 MOD 9) can be assigned one more element than their fair share. Starting with column 1, processors 1, 2, and 3 qualify for assignment to its remaining elements (these processors constitute list A for this column). The quota of all three processors is zero. However, the minimum of *R* (which is 5) and the number of processors with a zero quota (which is 3) is three. Since this value plus the quota of the processors in set A (which is zero) is greater than the number of unassigned elements in column 1, the heuristic proceeds to assign its elements. The first unassigned element of this column is an element of row 10; since none of the processors in set A appears in this row, we assign the element to one of the processors in set A, say processor 1. Since the quota of processor one is now -1, it is removed from set A. The same procedure is repeated for the second unassigned element of this column. For column 2, processors 1, 2, and 3 qualify again (these processors constitute set A). However, the quota of processors 1 and 2 is now -1, so they are removed from set A (set A now consists of processor 3). The minimum of *R* (which is three) and the number of processors with a zero quota in set A (which is one) is one. This value plus the quota of processor 3 (which is zero) is less than the number of unassigned elements of this slice, so heuristic continues with the next slice in the sorted list. Figure 6.17 presents the assignment of elements at the end of Phase 3. Column 2 along with rows 10 and 11 remain unassigned because the quota of the qualifying processors was less than the number of unassigned elements of those slices.

During Phase 4, the heuristic assigns the remaining unassigned elements. It initializes set A to the processors whose quota is not -1. If *R* is zero, it removes all processors with a zero quota from set A. It takes the first unassigned element and compares each processor in set A to the processors in the *K* slices that include this element. If a single processor in set A appears in all *K* slices, this element is assigned to it. Otherwise, we determine a processor in set A that appears in a slice corresponding to a dimension with the highest frequency of access and assign it to this element. If no processor in set A appears in any of the *K* slices, the

1	1	4	4	7	7	1
1	1	4	4	7	7	1
1	1	4	4	7	7	4
2	2	5	5	8	8	2
2	2	5	5	8	8	2
2	2	5	5	8	8	5
3	3	6	6	9	9	3
3	3	6	6	9	9	3
3	3	6	6	9	9	6
1		4	4	7	7	3
2		5	5	8	8	6

Figure 6.17

element is assigned to the processor in set A with the highest quota.

In our example, processors 3, 6, 7, 8, 9 will constitute the original set A. However, since the value of R is zero, all processors whose quota has been reached are removed from this set. Set A now consists of processor 9, and the remaining two elements are assigned to this processor. The final assignment of elements to processors is presented in Figure 6.18. Assuming a uniform distribution of access to the tuples of the relation, the assignment in Figure 6.18 causes the MDPS to use an average of 3.6 processors to execute the queries.

STEP 4:

We now remove the first constraint and consider the case where $\prod_{i=1}^{K} M_i \neq P$. In this case, we replace the original values of M_i with new values that do satisfy the first constraint. The new values of M_i should satisfy the following two constraints: 1) $\prod_{i=1}^{K} M_i = P$, and 2) the new values should "closely approximate" the original values of M_i. In order to achieve this, all possible divisors of P consisting of K terms $(T_1, T_2, ..., T_K)$ are computed (recall that K is the

1	1	4	4	7	7	1
1	1	4	4	7	7	1
1	1	4	4	7	7	4
2	2	5	5	8	8	2
2	2	5	5	8	8	2
2	2	5	5	8	8	5
3	3	6	6	9	9	3
3	3	6	6	9	9	3
3	3	6	6	9	9	6
1	9	4	4	7	7	3
2	9	5	5	8	8	6

Figure 6.18

dimensionality of the grid directory). Each term T_i of a divisor is to replace M_i. Recall that M_i represents the number of distinct processors that should appear in a given slice of dimension D_i. Any divisor with a term T_i whose value is greater than the number of elements in each slice of dimension D_i is eliminated because it would be impossible to have T_i unique processors assigned to that slice. From the remaining divisors, a divisor with a minimum value for the following equation is selected for the new value of M_i ($|M_i - T_i|$ represents the absolute difference between these two variables) :

$$\sum_{i=1}^{K} (FreqD_i * |M_i - T_i|) \tag{8}$$

By taking the frequency of access of each dimension into consideration, the heuristic will choose a divisor that closely approximates the value of M_i of the most frequently accessed dimension. Next, depending on whether N_i is a multiple of the new value of M_i, either Step 1, 2 or 3 is used to assign the elements to the processors.

Example 4:

Recall the parameters from Example 1 ($P = 9$, $N_1 = N_2 = 6$). The description of the problem in Example 1 required a single processor to execute each query; $M_1 = M_2 = 1$. Since $\prod_{i=1}^{K} M_i \neq 9$, the heuristic evaluates the possible divisors of 9 consisting of 2 terms; they are: (1,9), (3,3), and (9,1). The first and third divisors are eliminated because neither slice of the directory has 9 elements. The heuristic uses the value of 3 as the new value of M_1 and M_2 and proceeds to assign the elements to processors (Example 2 illustrates how this assignment is performed.) □

When none of the divisors of P qualify as the new values of M_i, the heuristic increments the value of P and proceeds to assign the elements using either Step 1, 2, or 3. It then distributes the elements assigned to the non-existing processor evenly among the existing P processors.

Example 5:

Recall the parameters from Example 1 ($N_1 = N_2 = 6$, $M_1 = M_2 = 1$). However, instead of a system consisting of 9 processors, assume one with 7 processors ($P = 7$). Since $\prod_{i=1}^{2} M_i \neq P$, the

Ticker_Symbol

PRICE	A-D	E-H	I-L	M-P	Q-T	U-Z
0 - 10	1	1	1	2	2	2
11 - 20	3	3	3	4	4	4
21 - 30	5	5	5	6	6	6
31 - 40	7	7	7	8	8	8
41 - 50						
51 - 60						

Figure 6.19

Ticker_Symbol

A-D	E-H	I-L	M-P	Q-T	U-Z
1	1	1	2	2	2
3	3	3	4	4	4
5	5	5	6	6	6
7	7	7			

Figure 6.20

possible divisors of 7 are determined; they are: (1, 7) and (7, 1). However, neither divisor qualifies as the new value of M_t because seven is greater than the number of elements in each slice of the directory. Consequently, P is incremented to 8. The new divisors of P are: (1, 8), (2, 4), (4, 2), and (8, 1). The first and last divisors are eliminated because each slice of a dimension has only 6 elements. The remaining two divisors qualify as the new value of M_t. Furthermore, the value of Equation 8 is the same for both divisors (the terms of either divisor are appropriate for the new values of M_t).

Assume we choose $M_1 = 2$ and $M_2 = 4$. While N_1 is a multiple of M_1, N_2 is not a multiple of M_2. The heuristic decrements N_2 until it is an even multiple of M_2 ($NewN_t = 4$). This new 6x4 grid is broken into 8 smaller 3x1 grids and each grid is assigned to a unique processor, resulting in the assignment of Figure 6.19. Before assigning the remaining elements, the heuristic analyzes the existing assignment and deletes the assignments to the fictitious processor 8 (resulting in the assignment of Figure 6.20). These elements along with the remaining elements are distributed across the original number of (7) processors using the heuristic described in Step 3. The final assignment of elements is presented in Figure 6.21. With this assignment, the MDPS employs an average of 3.3 processors while a one-dimensional

		Ticker_Symbol					
		A-D	E-H	I-L	M-P	Q-T	U-Z
P R I C E	0 - 10	1	1	1	2	2	2
	11 - 20	3	3	3	4	4	4
	21 - 30	5	5	5	6	6	6
	31 - 40	7	7	7	7	1	1
	41 - 50	3	3	1	2	2	6
	51 - 60	5	5	7	4	4	6

Figure 6.21

declustering strategy uses an average of 4 processors. □

In each of the examples in this section, we used a small directory to keep the discussion simple (the ratio of elements to processors was 4). However, in real applications, we imagine grid directories with thousands of slices and systems with 100 to 1000s of processors. Thus, the savings provided by the MDPS will be significantly higher than a fraction of a single processor (as we will demonstrate in Section 6.5.1). However, the heuristic does not always result in an assignment that obtains a better degree of parallelism than one of the one-dimensional declustering strategies. Consequently, the final assignment of elements to processors must always be evaluated to determine whether it is a better approximation of the ideal degree of parallelism than a one-dimensional declustering strategy. If not, then one might either reduce the number of attributes used to decluster the relation and reconstruct the grid directory or use the one-dimensional version of the MDPS.

Uniform Distribution of Tuples Across Processors

One simplifying assumption that we have made is that each element of the grid directory contains the same number of tuples. However, this is not a realistic assumption because a split along a dimension generally does not distribute the tuples evenly across the elements. For example, the split along the *price* attribute in Figure 6.5 resulted in a directory consisting of 4 elements. Two of these elements contain 2 tuples, one element has a single tuple and the remaining element has no tuples. Although this is an extreme example, [NIEV84] reports that on average a bucket is 70% full (assuming no correlation between the attributes). However, the MDPS does not assign buckets to processors; rather, it assigns the elements of the grid directory to processors. Furthermore, two or more elements of the grid directory might end up pointing to the same bucket (see Figure 6.5). In our experiment (presented in Section 6.5.1), we observed that on average the cardinality of each element is 50% of the cardinality of a fragment.

One of the original objectives of the MDPS was to uniformly distribute the tuples of a relation across the processors. In order to achieve this objective, after all the elements are

assigned to the processors, the heuristic analyzes the final assignment to determine if changing the assignment of elements to processors could reduce the difference in the number of tuples between the processor with most tuples (heaviest processor) and the one with fewest tuples (lightest processor). This is performed as follows. First, the heuristic analyzes the current assignment of elements and computes the number of tuples assigned to each processor. It then determines the element with the fewest tuples among those assigned to the lightest processor, and it determines the element with the most tuples among those assigned to the heaviest processor. The assignment of these two elements cannot simply be interchanged, because this will almost certainly destroy the original effort to closely approximate the value of M_i for each dimension D_i. However, two slices of a given dimension i can be interchanged without affecting the number of unique processors that appears in each. For example, two rows of Figure 6.1 can be interchanged without affecting the number of unique processors for the *price* and *ticker_symbol* attribute domains. In order to swap the assignment of two elements, the slices in each dimension that include these elements as members should be interchanged (a total of K interchanges for a K dimensional directory). For example, in order to interchange the assignment of element (1,4) with that of (3,6) in Figure 6.1, the assignment of column 4 should be swapped with that of column 6, and that of row 1 with row 3. However, each swap affects the weight of several processors. It might even change the identity of the lightest and heaviest processors. Furthermore, an intermediate assignment after a single swap might result in a better weight distribution than the final K swaps. Consequently, it is not appropriate to swap the processor assigned to the lightest element with that of the heaviest element. Rather, the heuristic swaps two slices along a single dimension.

Given the position of the lightest and heaviest element, there are at most K and at least one possible way of changing the two slices that include these elements. There are at most K because two slices of each dimension D_i can be interchanged and the directory is K dimensional. However, if both elements belong to a single slice of a dimension, swapping a slice with itself will have no impact on the weight difference. Furthermore, two elements cannot be a member of all K slices; thus, there must at least be one possible way of swapping two slices.

For example, in Figure 6.1, there are two possible ways of swapping element (1,4) with the element (3,6); either swap rows 1 and 3 or columns 4 and 6. However, if the second element is (1,6), the only alternative is to swap column 4 with column 6.

The search space of this minimization problem can be viewed as follows. Each node in the search space represents the current assignment of elements to the processors. The branching factor of each node is at most K and at least one. Each branch swaps the assignment of processors in two slices of dimension D_i. After each swap, the identity of the lightest and heaviest processor might change. However, the goal of reducing the weight difference between the heaviest and lightest processor remains (regardless of the identity of the processors). The optimal solution corresponds to a node with the minimum weight difference. Using the current assignment of the directory as the starting node, the heuristic uses a *Hill Climbing* [RICH83] procedure to search the space. This is achieved by always following the branch (out of one to K branches) that results in the minimum weight difference. If all branches increase the weight difference (or have no impact on it), the heuristic randomly chooses two slices of a randomly chosen dimension and swaps them (even if the swap increases the weight difference). This is because we assume that the heuristic has reached a *local minimum* in the search space and taking a random jump in the search space is one of the best ways of moving out of a local minimum [RICH83]. The heuristic then restarts with hill climbing using the resulting assignment of the random swap as a new node in the search space. It keeps track of the directory with the least weight difference while searching the solution space.

One might either require the heuristic to search the solution space until the weight difference is within a well defined threshold or limit the heuristic to visit a fixed number of nodes in the search space. The first alternative is not appropriate because if the defined threshold is infeasible, the heuristic will indefinitely visit the nodes in the search space and will never return. The second alternative is more appropriate because it ensures that the heuristic will return with a solution after a well defined interval of time. However, an even better alternative is a combination of these two where both the threshold and the number of nodes that the heuristic is allowed to visit are defined. This allows the heuristic to return either as soon as it

finds a reasonable solution or with the best solution it has found after visiting a preset number of nodes.

After the elements of the directory are assigned, and the difference in weight between the heaviest and lightest processor is minimized, the MDPS scans the relation a second time and assigns tuples to processors based on the assignment of the range of the partitioning attribute values in the grid directory. The grid directory is then stored in the database catalog and used by the optimizer to localize the execution of the queries accessing this relation.

We have implemented a prototype of the MDPS, along with the heuristic for assigning the elements and the heuristic for approximating a uniform distribution of tuples across the processors. In the next section, we compare the degree of parallelism obtained by the heuristic with the theoretical minimum, and evaluate how closely we can approximate a uniform distribution of tuples across the processors.

6.5.1. Performance of Heuristic for Assigning Entries to Processors

In the case of $|F| > P$, the assignment of slices to processors can be posed as a minimization problem where the goal is to minimize the average number of employed processors to one. The theoretical lower bound of this problem is shown in [GHAN90c] to be:

$$\frac{P * \left\lceil K * \left(\frac{\prod_{i=1}^{K} N_i}{P} \right)^{\frac{1}{K}} \right\rceil}{\sum_{i=1}^{K} N_i}$$

This lower bound does not necessarily guarantee the existence of a solution. However, we can evaluate the heuristic by comparing the number of processors that it employs with this bound (i.e., a pessimistic evaluation).

For the purposes of this evaluation, we used a 100,000 tuple relation from the Wisconsin Benchmark [BITT83]. We used the unique1 and unique2 attributes as the declustering attributes and assumed that one is the optimal number of processors that should be employed to execute the queries accessing either attribute ($M_1 = M_2 = 1$). The capacity of each bucket was

200 tuples. We analyzed the performance of the heuristic for two different frequencies of access to each attribute. While the first experiment assumes a 50% frequency of access to each attribute, the second experiment assumes an 80%/20% frequency of access.

When each attribute has a 50% frequency of access, the MDPS uses a cyclic splitting strategy to create the grid directory. With these parameters, the MDPS created a 32x31 directory on the relation. Thus, on average each element contains 100.81 tuples (50.4% of the cardinality of a bucket). Table 6.1 presents the theoretical lower bound, the average number of processors employed by the heuristic, and the percentage difference between these two values as a function of the number of processors (P) in the system. The results reveal that the heuristic approximates the theoretical lower bound with a maximum error of 16% here.

In the remaining two columns of Table 6.1, we present the number of processors that would be used by a one-dimensional declustering strategy and the number of processors freed by the MDPS (using the heuristic) to perform useful work. The results reveal that the total number of freed processors increases as a function of the number of processors in the system.

Table 6.2 presents the percentage weight difference between the lightest and heaviest processor after the initial assignment of elements, and that of the best solution after the heuristic

Table 6.1
Two Dimensional Declustering
of a 100,000 Tuple Relation (50% Freq of Access)

P	Theoretical Lower bound	Heuristic for MDPS, Average Number of processors used	% diff	One-Dim Declust, Average Number of processors used	Number of processors freed by the MDPS
8	2.92	3.13	7.17%	4.5	1.35
10	3.18	3.63	14.35%	5.5	1.87
16	4.06	4.26	4.84%	8.5	4.24
20	4.76	4.76	0.00%	10.5	5.74
32	6.10	6.39	4.84%	16.5	10.11
64	8.13	8.52	4.84%	32.5	23.98
128	12.19	12.39	1.64%	64.5	52.11
256	16.25	16.26	0.04%	128.5	112.24

Table 6.2
% weight difference between lightest and heaviest processor of the best found assignment(50% Freq of Access)

P	% weight diff before load balancing	% weight diff after visiting 100 nodes	% weight diff after visiting 500 nodes	% weight diff after visiting 1000 nodes
8	19.83%	5.59%	5.42%	5.42%
10	27.68%	5.24%	3.94%	3.94%
16	24.93%	18.88%	5.69%	5.69%
20	34.11%	7.25%	5.92%	5.92%
32	37.49%	16.51%	13.53%	13.41%
64	62.18%	16.41%	16.41%	16.14%
128	105.28%	35.55%	35.55%	30.00%
256	160.44%	34.88%	34.88%	33.24%

has visited 100, 500, and 1000 nodes in the search space. A general trend in this table is that as the number of processors in the system increases, the percentage weight difference between the lightest and heaviest processor also increases. This is because there are 992 elements in the directory and, as the number of processors increases, the number of elements per processor decreases (e.g., when $P = 256$, each processor contains 3 or 4 elements). Thus, a slight weight difference between two elements assigned to two different processors can result in a large percentage difference between the lightest and heaviest processor. A second important observation from these results is that the performance of the hill climbing search technique is very good because the weight difference is significantly reduced after visiting only 100 nodes. In addition, for most values of P in Table 6.2, visiting more than 100 nodes reduces the weight difference only marginally. In summary, the heuristic does a very good job of uniformly distributing the tuples across the processors.

In a second experiment, we assumed a skewed distribution of access, with an 80% frequency of access for the unique1 attribute and a 20% frequency of access for the unique2 attribute. In this case, the MDPS splits the directory four times more frequently on the unique1 attribute than the unique2 attribute, creating a 65x16 directory. Each element on average contains 96.15 tuples (48.08% of a bucket). Table 6.3 presents the performance of the heuristic for this case. The results indicate that the heuristic approximates the optimal

Table 6.3
Two Dimensional Declustering
of 100,000 Tuple Relation (80%/20% Freq of Access)

P	Theoretical Lower bound	Heuristic for MDPS, Average Number of processors used	% diff	One-Dim Declust, Average Number of processors used	Number of processors freed by the MDPS
8	2.27	2.47	8.73%	2.40	-0.07
10	2.59	2.60	0.29%	2.80	0.20
16	3.36	3.37	0.36%	4.00	0.63
20	3.70	3.72	0.44%	4.80	1.08
32	4.74	4.95	4.41%	7.20	2.25
64	7.11	7.23	1.67%	13.60	6.37
128	9.48	9.70	2.31%	26.40	16.70
256	15.80	16.04	1.50%	52.00	35.96

solution with a maximum margin of error of 10% in this case.

Table 6.3 also contains the number of processors used by a one-dimensional declustering strategy when the most frequently accessed attribute was used as the partitioning attribute. When the system is composed of 8 processors, the one-dimensional declustering strategy employs fewer processors than the heuristic solution because the total number of processors is so low that the savings provided by a two dimensional declustering strategy are marginal. However, as the number of processors is increased, the heuristic uses fewer processors than a one-dimensional declustering strategy.

Table 6.4 presents the performance of the heuristic for approximating a uniform distribution of tuples across the processors. Again, the results reveal that the heuristic reduces the weight difference between the lightest and heaviest processors.

The results presented here reveal that: 1) the number of processors employed by the MDPS closely approximates the optimal solution, 2) given a system with a large number of processors, the number of processors used by the MDPS is significantly closer to the optimal number of processors than a one-dimensional declustering strategy, and 3) by swapping the assignment of different slices using a hill climbing search technique, the MDPS can closely

Table 6.4
% weight difference between lightest and heaviest processor of the best found assignment(80%/20% Freq of Access)

P	% weight diff before load balancing	% weight diff after visiting 100 nodes	% weight diff after visiting 500 nodes	% weight diff after visiting 1000 nodes
8	7.93%	1.86%	1.24%	0.87%
10	7.49%	2.08%	2.08%	1.60%
16	12.12%	5.39%	3.11%	2.62%
20	9.30%	4.92%	4.92%	2.35%
32	15.63%	7.37%	5.51%	5.51%
64	32.23%	12.14%	10.37%	10.37%
128	44.83%	20.83%	20.33%	19.62%
256	60.12%	39.43%	37.36%	37.36%

approximate a uniform distribution of tuples across the processors in a short interval of time.

6.6. Other Advantages of MDPS

6.6.1. Support for Small Relations

In database machines with hundreds or thousands of processors, relations with low cardinalities must be partially declustered across a subset of processors [COPE88]. The MDPS does a very good job at supporting small relations since the number of fragments created by the MDPS is dependent on the processing capability of the system and the resource requirements of the queries in the workload and independent of the number of processors in the system. If the number of fragments of a relation is less than the number of processors, then the relation will automatically be partitioned across a subset of the processors.

6.6.2. Support for Relations with Non-Uniform Distributions of the Partitioning Attribute Values

The MDPS is also capable of declustering relations with non-uniformly distributed partitioning attribute values since the cardinality of each fragment is not based on the partitioning attribute value. Once the MDPS determines the cardinality of each fragment, it will decluster a relation based on that value. For example, assume relation R is being declustered on only a

single attribute and that it has a cardinality of 100,000 tuples. In addition, assume that 4,000 of these tuples have 2 as their partitioning attribute value. If MDPS determines that the cardinality of each fragment should be 1000 tuples, the tuples with a partitioning attribute value of 2 will be distributed among 4-5 fragments and assigned to different processors. Thus, if a query with an exact match predicate for value 2 is submitted to the system, the query will be directed to 4 or 5 processors (rather than one processor, had the relation been declustered using either the hash or range partitioning strategy).

In summary, in this chapter we have presented the design of a multidimensional partitioning strategy, described how a grid directory is constructed on a relation, and provided heuristics for: 1) assigning the elements of the directory to processors, and 2) approximating a uniform distribution of tuples across the processors. We evaluated the heuristic for assigning the elements by comparing its performance with the theoretical lower bound found in [GHAN90c]. The results indicate that the MDPS closely approximates the ideal number of processors that should execute the queries in the workload while resulting in a close approximation of a uniform distribution of tuples across the processors.

CHAPTER 7

PERFORMANCE EVALUATION OF ONE DIMENSIONAL MDPS

In this chapter, the performance of a one-dimensional version of MDPS is compared with the range and hash declustering strategies. This evaluation serves two purposes: 1) it demonstrates that the MDPS is a better declustering strategy than the range and hash declustering strategies, and 2) it shows the correctness of the criteria used by the MDPS to create the fragments of a relation (outlined in Section 1.1 of Chapter 6). The rest of this chapter is organized as follows. In the next section we present and motivate the queries used for this evaluation. In Section 2, the performance of the alternative declustering strategies for the different queries is presented and analyzed.

7.1. Workload Definition

The workload used to compare the performance of the MDPS with the other declustering strategies is primarily the same as that described in Chapter 4. However, we used a different set of queries for this performance evaluation. In order to justify their selection consider Figure 7.1. In this figure, space A represents the domain of all possible queries. The region marked "Range" represents the region in space A for which the range partitioning strategy provides the best response time. This region consists of queries whose execution requires minimal CPU and I/O resources since the range partitioning strategy localizes their execution to a single processor and avoids the overhead associated with a multi-processor query.

The area marked "Hash" represents the query types for which the hash partitioning strategy provides the best response time possible. Ignoring the shaded area for now, this region consists of queries with high resource requirements since the hash partitioning strategy directs such queries to all the processors, thus utilizing intra-query parallelism effectively to obtain

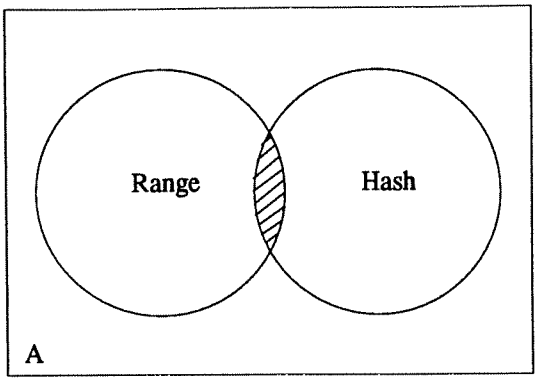


Figure 7.1

the best possible response time.

These two regions (again ignoring the shaded area) are disjoint since the range partitioning strategy does not perform as well as the hash partitioning strategy for queries with high resource requirements and, similarly, the hash partitioning strategy does not perform as well as the range partitioning strategy for queries with minimal resource requirements.

The shaded region represents those selection queries with an equality predicate on the partitioning attribute that have minimal resource requirements. Both the range and hash partitioning strategies can localize the execution of such queries to a single processor.

In order to demonstrate that the MDPS is generally a better declustering strategy, we must show that it covers a region in space A that largely subsumes both regions marked "Range" and "Hash". Note that we do not have to consider all the different queries that might occur in either the "Range" or the "Hash" region in order to achieve this objective. Rather, we must demonstrate that the MDPS performs equally as well as:

- the range declustering strategy for queries with low resource requirements
- the hash partitioning strategy for queries with high resource requirements
- both partitioning strategies for queries with an equality predicate and minimal resource requirements.

Finally, in those cases where the mix of queries have conflicting partitioning requirements, we must demonstrate that the MDPS provides better performance than both the hash and range

partitioning strategies.

For a query with minimal resource requirements, we used a 0.001% range selection on a 1 million tuple relation using a clustered index. This query retrieves and processes 10 tuples. Its single processor, single user execution time is 0.08 seconds on Gamma. For a query with high CPU and I/O requirements, we selected a 10% range selection on a 1 million tuple relation using a clustered index. This query retrieves and processes 100,000 tuples and has an execution time of 54.33 seconds. The query with an equality predicate and minimal resource requirements is redundant since it is a special case of the first class of queries (0.001% selection query). As long as the MDPS performs as well or better than the range partitioning strategy for the 0.001% range selection query, it has automatically satisfied the execution requirements of queries with an equality predicate and minimal resource requirements.

In order to demonstrate the superiority of the MDPS in those cases where the query mix contains queries that have conflicting partitioning requirements, we chose a workload consisting of a mix of the 0.001% and 10% selection queries. Note that while the execution of the 0.001% selection must be localized to a single processor to minimize its execution time, the 10% selection should be executed by all the processors. Two cases were considered. In the first, we used an equal mix of both query types and varied the multiprogramming level. In the second, we fixed the multiprogramming level at 50 and varied the mix of the two query types. As we will demonstrate in Section 2 of this chapter, the MDPS provides superior performance in such situations.

7.2. Performance of Alternative Declustering Strategies

In this section, we evaluate the performance of the alternative declustering strategies using the queries described in the previous section. The alternative partitioning strategies declustered the relations across all the processors. The range and hash partitioning strategies distributed the tuples of each relation uniformly across the 24 processors, while the MDPS resulted in an approximately uniform distribution of the tuples across the processors. The execution paradigm of the alternative partitioning strategies and the physical configuration of

the system for running the experiments are the same as those described in the introduction of Chapter 5 (see Figure 5.1).

We measured *CostOfPart*, the overhead of using each additional processor to execute a query, to be 26 milliseconds in Gamma. Since operators are scheduled and terminated sequentially, the total overhead is a linear function of the number of participating processors.

Since the relation is declustered on a single attribute, the MDPS creates a one dimensional grid directory that is identical to a range table created by the range partitioning strategy. For both the range and the multidimensional partitioning strategies, the optimizer utilizes a binary search algorithm to search the range-table to determine which processors should participate in the execution of a query. We measured the overhead of searching the range table and initializing the query packet (i.e., CostOfSearch) to be 0.243 milliseconds per range table entry. These parameters are used by the MDPS to determine the cardinality of each fragment for a given relation.

In the following sections, when we refer to the performance of a specific partitioning strategy for a particular type of query, we are implying that the selection predicate of the query is applied to the partitioning attribute.

7.2.1. 0.001% Selection Using a Clustered Index

In Figure 7.2, the throughput as a function of the multiprogramming level of a 0.001% selection query using a clustered index is presented for each of the alternative declustering strategies.

The execution time of this query using a single processor is 0.08 seconds. Using equation 4 of Chapter 6 (and the values of CostOfSearch and CostOfPart from the previous section), the value of M for this query was calculated to be 0.057 and hence only a single processor should be used. The MDPS will decluster each million tuple relation into 5,700 (0.057 * 1,000,000) fragments with about 175 tuples per fragment. Since there are more fragments than processors, the fragments are distributed in a round-robin fashion among the processors.

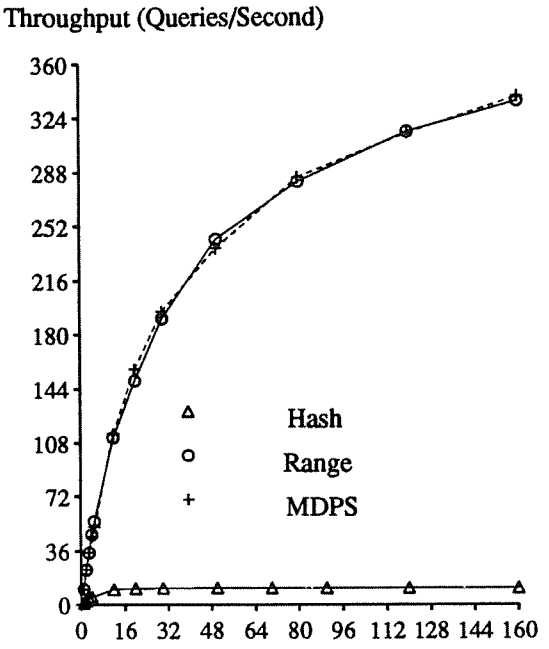


Figure 7.2: 0.001% Indexed Selection

While the optimizer has enough information for the multidimensional and range declustering strategies to localize the execution of this query to 1 or 2 processors, the hash partitioning strategy must direct this query to all the processors because it involves a range predicate. Consequently, at low multiprogramming levels, the hash partitioning strategy incurs the overhead associated with a multi-processor query and results in a lower performance than that of the range and multidimensional declustering strategies. At a multiprogramming level of 20, with the hash partitioning strategy, the CPU of each individual processor becomes 100% utilized causing the throughput of the system to level off. With the range and multidimensional partitioning strategies, the throughput of the system increases at higher multiprogramming levels as the idle resources become utilized by the additional concurrently executing queries.

The throughput with the range and multidimensional partitioning strategies is almost identical at all multiprogramming levels (the maximum difference is less than 2.5%) because both partitioning strategies localize the execution of the query to a single processor most of the time. While the multidimensional partitioning strategy must search a significantly larger direc-

tory (5700 instead of 24 entries), the execution time of the query is high enough to render this overhead insignificant (and in addition, this overhead was taken into consideration for creating the grid directory).

As these results demonstrate, the multidimensional partitioning strategy performs as well as the range partitioning strategy for queries with minimal resource requirements.

7.2.2. 10% Selection Using a Clustered Index

The throughput of the system for a 10% selection using a clustered index for the alternative declustering strategies is presented in Figure 7.3. Using equation 4 of Chapter 5, the value of M is 89.5 for this query and the MDPS declusters each million tuple relation into 895 fragments. Ideally, 90 processors should be used to execute this query, but since only 24 processors are available, each processor must perform approximately 3.7 (i.e., $\frac{90}{24}$) times the optimal amount of work. The MDPS directs this query to all the processors.

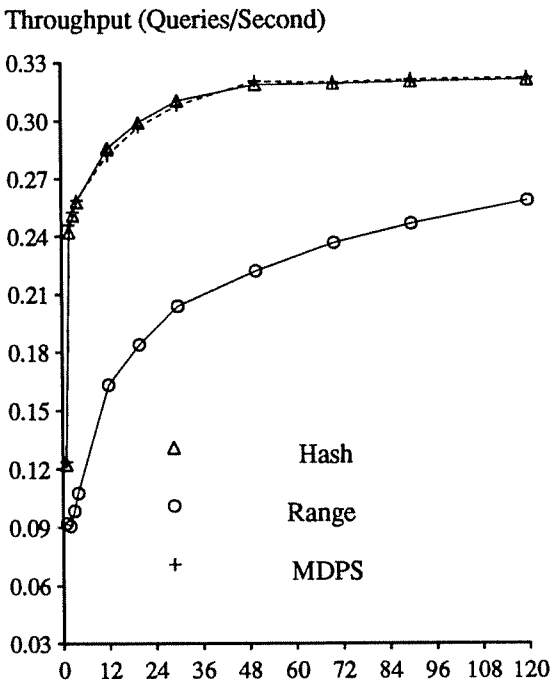


Figure 7.3: 10% Indexed Selection

The range partitioning strategy declusters each relation into 24 fragments, each containing a distinct range of the partitioning attribute values. Since the range of a 10% selection query will overlap at most the range of 3 fragments, it will use at most 3 processors to execute this query.

At a multiprogramming level of one, the throughput with the hash and multidimensional declustering strategies is 16% higher than that of the range partitioning strategy because the hash and multidimensional declustering strategies utilize intra-query parallelism effectively while the range partitioning strategy directs the query to the absolute minimum number of processors and performs most of the work in a sequential manner.

With the range partitioning strategy, the throughput of the system increases at higher multiprogramming levels as previously idle resources become utilized by the additional concurrently executing queries. For the hash and multidimensional partitioning strategies, the throughput of the system increases due to processor sharing and utilization of the elevator algorithm at the disk controller. These declustering strategies utilize parallelism more effectively, thus outperform the range declustering strategy.

The 10% range selection query represents the best case scenario for the hash partitioning strategy and the worst case scenario for the range partitioning strategy, as the hash partitioning strategy utilizes intra-query parallelism effectively to produce the best response time and throughput. The multidimensional declustering strategy performs as well as the hash partitioning strategy because it also utilizes a high degree of parallelism to execute this query.

7.2.3. A Mixed Workload

For our final experiment, we used a mixed workload consisting of one-half 0.001% selection queries and one-half 10% selection queries. **It is important to note that the partitioning requirements of these two queries conflict with one another.** While the 0.001% selection query should be directed to a single processor, the response time of the 10% selection query will be minimized if all the processors are used. While neither the range nor hash partitioning strategies can resolve the conflicting demands of these two queries, the MDPS can. The performance of the

different declustering strategies for this mix of queries is presented in Figure 7.4.

The MDPS declusters each relation into 1688 fragments (using equations 5 and 6 of Chapter 6) which were distributed among the processors in a round-robin fashion. Thus, the optimizer has enough information with this partitioning strategy to direct the 0.001% selection query to a single processor while using all the processors to execute the 10% selection query. On the other hand, the hash partitioning strategy uses all the processors for both queries. The range partitioning strategy directs the execution of the 0.001% selection query to a single processor and uses at most 3 processors to execute the 10% selection query.

The hash and multidimensional partitioning strategies outperform the range partitioning strategy because the execution of the 10% selection dominates the computation of the average throughput. That is, both queries occur with equal frequency in this experiment, and the response time of the 10% query is significantly higher than that of the 0.001% selection query.

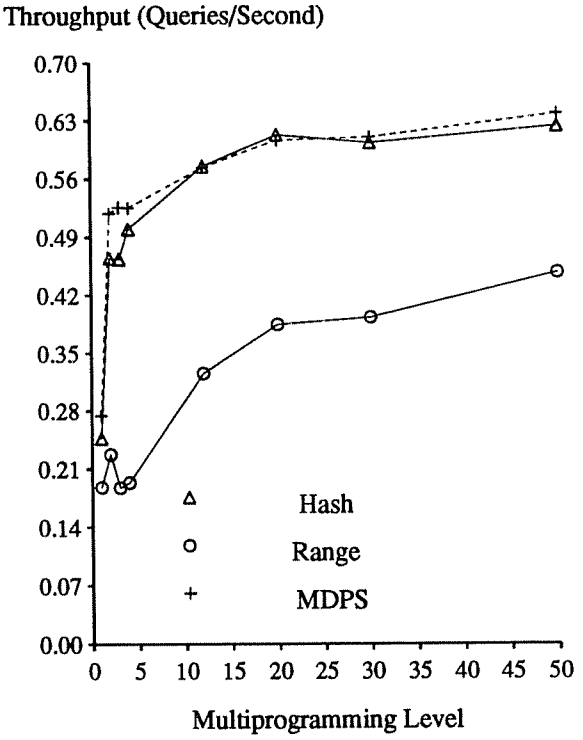


Figure 7.4: Mixed Workload

so those partitioning strategies that maximize throughput for the 10% selection query also maximize the throughput for the whole workload. This also explains why the throughput of the hash and multidimensional declustering strategies converge at multiprogramming levels higher than four. The impact of the 10% selection query on the workload is so significant that the savings provided by the MDPS for the 0.001% selection query is not significant.

We can generalize on the results obtained for this mix of queries and speculate on workloads consisting of different mixes of queries. As the frequency of occurrence of queries with minimal resource requirements (e.g., the 0.001% selection) is increased, the throughput of the range and multidimensional partitioning strategies converge and outperform the hash partitioning strategy. Conversely, as the frequency of queries with high resource requirements (e.g., the 10% selection query) is increased, the throughput of the hash and multidimensional partitioning strategies converge and outperform the range partitioning strategy.

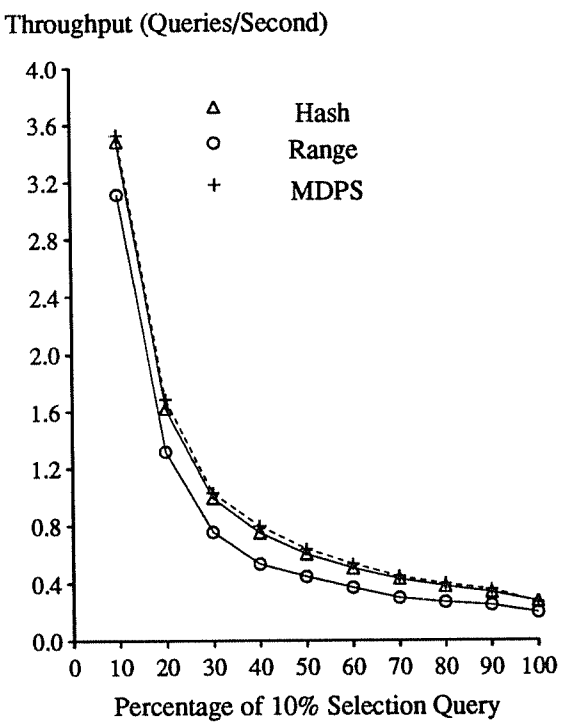


Figure 7.5: Multiprogramming Level = 50

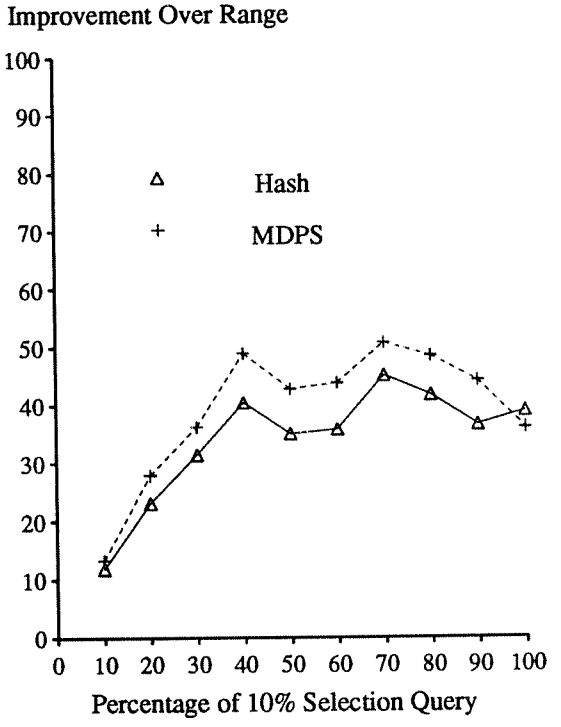


Figure 7.6: Multiprogramming Level = 50

To support this hypothesis, we fixed the multiprogramming level of the system at 50 and varied the mix of 10% and 0.001% selection queries in the workload. The results are presented in Figure 7.5. In this figure, the X axis represents the percentage of 10% selection queries in the workload. For example, at point 80 on the X axis, eighty percent of queries in the workload were 10% selection queries and the remaining twenty percent were 0.001% selection queries. The performance of the hash and multidimensional partitioning strategies are almost identical when the 10% selection query constitutes more than ten percent of the workload. The MDPS outperforms the hash partitioning strategy by a slight margin since it localizes the execution of the 0.001% selection queries to a single processor.

In Figure 7.6, we present the percentage improvement in throughput provided by the hash and multidimensional partitioning strategies relative to the range partitioning strategy. As the percentage of the 10% selection queries is increased, the hash and multidimensional partitioning strategies outperform the range partitioning strategy by an increasingly widening margin. The percentage improvement levels off when the 10% selection query constitutes forty percent of the queries in the workload because the disk has become 100% utilized at this point.

In Figure 7.7, we present the throughput of the alternative partitioning strategies at a multiprogramming level of fifty for a variety of workloads consisting of a very low percentage of 10% selection queries. (This figure is just an enlargement of the lower limit not shown in Figure 7.5.) In this figure, the range partitioning strategy begins to outperform the hash partitioning strategy when the 10% selection query constitutes less than five percent of the queries in the workload because the range partitioning strategy can localize the execution of the 0.001% selection queries to a single processor (and these queries now constitute more than 95% of the queries in the workload).

The MDPS outperforms both the range and hash partitioning strategies by close to thirty percent when the 10% selection query constitutes five to seven percent of the queries in the workload. For this range of workloads, neither the range nor the hash partitioning strategy is

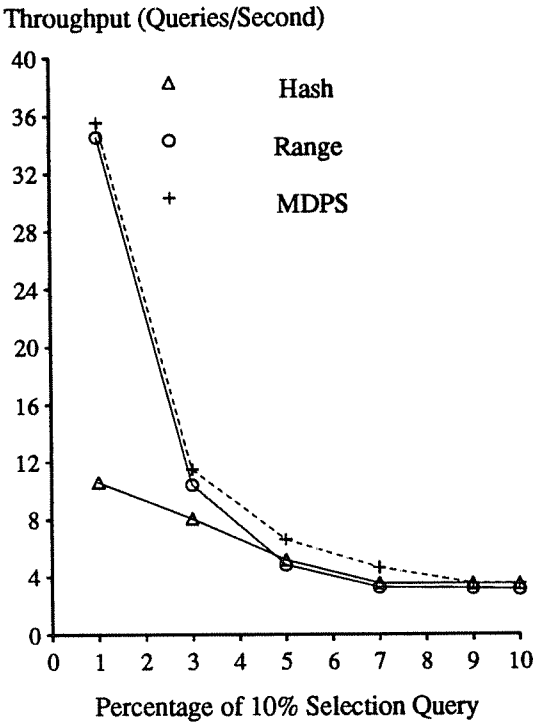


Figure 7.7: Multiprogramming Level = 50

appropriate. Since the MDPS provides the appropriate execution paradigm for both queries in the workload, it can outperform the other two partitioning strategies.

In summary, in this chapter we demonstrated that the one-dimensional version of MDPS is generally a better declustering strategy than either the range or hash declustering strategy. This is because the MDPS utilizes the characteristics of the queries to create the fragments of the relation.

CHAPTER 8

SUMMARY

8.1. Conclusions

In this dissertation, we have characterized the performance of the alternative declustering strategies for different types of selection queries. Briefly, we showed that the execution of queries with minimal resource requirements should be localized to a few processors in order to avoid the overhead associated with parallelism. On the other hand, a high degree of parallelism should be employed for queries with high resource requirements in order to: 1) reduce the response time of the queries, and 2) avoid the formation of hot spots and bottleneck processors (i.e., to achieve good load balancing). While the hash partitioning strategy localizes the execution of exact match queries, the range partitioning strategy attempts to localize the execution of all query types regardless of their selectivity factor.

We also presented the design of a new multidimensional partitioning strategy. This partitioning strategy declusters a relation based on: 1) the resource requirements of the queries that constitute the workload and, 2) the processing capability of the system. Furthermore, it can decluster the relation on multiple attributes in order to localize the execution of a greater variety of queries, including queries with predicates that access different attributes. We compared the performance of the one-dimensional version of MDPS with the range and hash declustering strategies and demonstrated its superiority.

8.2. Future Research Directions

Physical database design in multiprocessor database machines is a new and emerging area of research. This dissertation is a cornerstone for evaluating the alternative physical organizations and their impact on the performance of the system. We plan to extend both the

performance study and the design of the multidimensional declustering strategy in several ways. We consider each topic individually.

8.2.1. Performance of the Alternative Declustering Strategies

In this dissertation, the performance of the alternative declustering strategies was evaluated for selection queries only. We plan to extend this evaluation and study the impact of update queries and skewed distributions of the partitioning attribute values on the performance of the alternative declustering strategies.

Update Queries

The primary reason for analyzing update queries is to determine the cost associated with preserving the partitioning constraint of the hash, range, and multidimensional partitioning strategies. When an update query modifies the value of the partitioning attribute of a tuple, that tuple may migrate to another processor in order to preserve the integrity of the partitioning constraint. We are interested in analyzing the impact of this on the performance of the alternative declustering strategies.

In addition, update queries provide the means for analyzing the impact of different concurrency control algorithms and recovery methods on the throughput of a multiprocessor database machine.

Skewed Distribution of the Partitioning Attribute Values

A major assumption of this performance study was a uniform distributions of the partitioning attribute values. We plan to analyze the impact of skewed distribution of the partitioning attribute values on the performance of the alternative partitioning strategies. The major reason for this is that the range and hash partitioning strategies cannot guarantee a uniform distribution of tuples across the processors, while the multidimensional partitioning strategy attempts to guarantee a uniform distribution. We are interested in analyzing the significance of this aspect of the multidimensional declustering strategy and its impact on the performance; especially compared to the other declustering strategies.

8.2.2. The Multidimensional Partitioning Strategy

The design of Multidimensional Partitioning Strategy (MDPS) itself also requires further investigation. We want to determine if the assignment of elements to processors is an NP-complete [GARE71] problem, and if so, introduce other heuristics which can result in a better approximations of the ideal degree of parallelism. If not, we are interested in polynomial-time algorithms that obtain the optimal assignment of entries to processors.

We intend to perform a sensitivity analysis of M (i.e., the ideal degree of parallelism determined based on the characteristics of the workload) to determine the impact of its margin of error on the performance of the system.

We described the design of the MDPS assuming the attributes used to decluster the relation are already defined. However, what fraction of processors should be employed by queries that access an attribute ($\frac{M_i}{P} \leq Threshold$) in order to qualify that attribute as one of the partitioning attributes? This is an interesting research topic that requires further investigation.

Another interesting research topic is dynamic file reorganization in the presence of update queries and changing workload characteristics. The MDPS statically declusters a relation based on the underlying physical characteristics of the relation and its workload definition. However, databases are not static; the queries accessing a given attribute might change and require a new degree of parallelism. In addition, update queries might result in an imbalance of tuples across the processors. By dynamically reorganizing the physical layout of relations, the MDPS should be able to respond to these changes. In the future, we plan to study dynamic on-line file reorganization and the related tradeoffs.

REFERENCES

[ALEX88] Alexander, W., et. al., "Process and Dataflow Control in Distributed Data-Intensive Systems," Proc. ACM SIGMOD Conf., Chicago, IL, June 1988.

[ANON84] Anon, et. al., "A Measure of Transaction Processing Power," Datamation 1984.

[ASTR76] Astrahan, M. M., et. al., "System R: A Relational Approach to Database Management," ACM Transactions on Database Systems, Vol. 1, No. 2, June, 1976.

[BORA84] Boral, H, and DeWitt, D. J., "A Methodology for Database System Performance Evaluation," Proceedings of the 1984 SIGMOD Conference, Boston, MA, June, 1984.

[BORA90] Boral, H., Alexander, W., Clay, L., Copeland G., Danforth, S., Franklin, M., Hart, B., Smith, M., Valduriez, P., "Prototyping Bubba, A Highly Parallel Database System" IEEE Transaction on Knowledge and Data Engineering, March, 1990.

[BITT83] Bitton D., D.J. DeWitt, and C. Turbyfill, "Benchmarking Database Systems - A Systematic Approach," Proceedings of the 1983 Very Large Database Conference, October, 1983.

[CHOU85a] Chou, H-T, DeWitt, D. J., Katz, R., and T. Klug, "Design and Implementation of the Wisconsin Storage System (WiSS)" Software Practices and Experience, Vol. 15, No. 10, October, 1985.

[CHOU85b] Chou, H-T, Dewitt, D. J., "An Evaluation of Buffer Management Strategies for Relational Database System," Proceedings of the 1985 Very Large Database Conference, October, 1985.

[COME79] Comer, D., "The Ubiquitous B-Tree" ACM Computing Surveys, Vol. 11, No. 2, June, 1979.

[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, Illinois, May 1988.

[DEWI84] DeWitt, D. J., Katz, R., Olken, F., Shapiro, D., Stonebraker, M. and D. Wood, "Implementation Techniques for Main Memory Database Systems," Proceedings of the 1984 SIGMOD Conference, Boston, MA, June, 1984.

[DEWI88] Dewitt, D., Ghandeharizadeh, S., and D. Schneider, "A Performance Analysis of the Gamma Database Machine", Proceedings of the 1988 SIGMOD Conference, Chicago, IL, June 1988.

[DEWI90] DeWitt, D., Ghandeharizadeh, S., Schneider, D., Bricker, A., Hsiao, H., R. Rasmussen, "The Gamma Database Machine Project," IEEE Transaction on Knowledge and Data Engineering, March, 1990.

[DU82] Du, H.C. and Sobolewski, J.S., "Disk Allocation for Cartesian Product Files on Multiple-Disk Systems," ACM Trans. Database Systems, Vol. 7, No. 1, March 1982, pp.

82-101.

[GHAN90a] Ghandeharizadeh, S., and D. J. DeWitt, "Performance Analysis of Alternative Declustering Strategies," Proceedings of the 6th International Conference on Data Engineering, Los Angeles, CA, February 1990.

[GHAN90b] Ghandeharizadeh, S., and D. J. DeWitt, "Factors Affecting the Performance of Multiuser Database Management Systems," Proceedings of the 1990 SIGMETRICS Conference, Boulder, Colorado, May 1990.

[GHAN90c] Ghandeharizadeh, S., Meyer, R.R., Schultz, G., Yackel, J., "A Class of Array Coloring Problems and a Multi-processor Database Application" in preparation.

[GRAE90] Graefe, G., "Volcano: An Extensible and Parallel Dataflow Query Processing System," Oregon Graduate center, Computer Science Technical Report, Beaverton, OR., June 1989.

[GARE71] Garey, M. R., Johnson, D. S., **Computers and Intractability: A Guide to the Theory of NP-Completeness**, New York, 1971.

[GRAY78] Gray, J. N., "Notes on Database Operating Systems," in Lecture Notes in Computer Science 60, Advanced course on Operating Systems, ed. G. Seegmuller, Springer Verlag, New York 1978.

[GUTT84] Guttman, A., "R-trees: A Dynamic Index Structure for Spatial Searching,"' Proc. ACM SIGMOD, 47-57, 1984.

[INTE88], Intel Corporation, **iPSC/2 User's Guide**, Intel Corporation Order No. 311532-002, March, 1988.

[JARK84] Jarke, M. and J. Koch, "Query Optimization in Database Systems," ACM Computing Surveys, Vol. 16, No. 2, June, 1984.

[KIM85] Kim, M., "Parallel Operation of Magnetic Disk Storage Devices: Synchronized Disk Interleaving," Proceedings of the Fourth International Workshop on Database Machines, Springer Verlag, March 1985.

[KIM88] Kim, M.H., and Pramanik, S., "Optimal File Distribution for Partial Match Retrieval," Proc. ACM-SIGMOD Conf., Chicago, IL., June 1988, pp. 173-182.

[LIVN87] Livny, M., Khoshafian, S., Boral, H., "Multi-Disk Management Algorithms," Proc. of the 1987 ACM SIGMETRICS Int'l Conf. on Measurement and Modeling of Computer Systems, May 1987.

[NIEV84] Nievergelt, J., Hinterberger, J., Sevcik, K.C [84]. "The Grid File: An Adaptable, Symmetric Multikey File Structure" ACM Transactions on Database Systems, 9, No. 1, March, 1984.

[PRAM89] Pramanik, S., and M. H. Kim, "Database Processing Models in Parallel Processing Systems," **Database Machines**, Boral, H., and P. Faudemay, eds., Springer-Verlag, 1989.

[RIES78] Ries, D. and R. Epstein, "Evaluation of Distribution Criteria for Distributed Database Systems," UCB/ERL Technical Report M78/22, UC Berkeley, May, 1978.

[RICH83] Rich, E., **Artificial Intelligence**, New York, 1973.

[ROBI81] Robinson, J. T., "The k-d-B-tree: A Search Structure for Large Multidimensional Dynamic Indexes," Proceedings of the 1981 SIGMOD Conference, New York, 1981.

[SELI79] Selinger,P. G., et. al., "Access Path Selection in a Relational Database Management System," Proceedings of the 1979 SIGMOD Conference, Boston, MA., May 1979.

[SELL87] Sellis, T., Roussopoulos, N., Faloutsos, C., "The R+ tree: a dynamic index for multi-dimensional objects,"' Proceedings of 13th Int. Conf. on VLDB, 1987.

[STON88] Stonebraker, M., Patterson, D., Ousterhout, J., "The Design of XPRS," Proceedings of the 1988 VLDB Conference, Los Angeles, Ca., Sept, 1988.

[TAND85] Tandem Computers Inc., "Enscribe Programming Manual," Tandem Part 82583-A00, March 1985.

[TAND88] Tandem Performance Group, "A Benchmark of Non-Stop SQL on the Debit Credit Transaction", Proceedings of the 1988 SIGMOD Conference, Chicago, IL., June 1988.

[TERA83] Teradata Corp., *DBC/1012 Data Base Computer Concepts & Facilities*, Teradata Corp. Document No. C02-0001-00, 1983.

[TERA85a] Teradata Corp., *DBC/1012 Data Base Computer System Manual, Rel. 2.0*, Teradata Corp. Document No. C10-0001-02, November 1985.

[TERA85b] Teradata Corp., *DBC/1012 Data Base Computer Reference Manual, Rel. 2.0*, Teradata Corp. Document No. C03-0001-02, November 1985.

[WAGN73] Wagner, R. E., "Indexing Design Considerations," IBM Sys. J., vol. 12, no. 4, pp. 351-367, Dec. 1973.

[YAO79] Yao, S. B., "Optimization of Query Evaluation Algorithms," ACM Trans. on Database Systems, pp. 133-155, June 1979.