MIT/LCS/TR-192

ATTRIBUTE PARTITIONING IN A SELF - ADAPTIVE RELATIONAL DATABASE SYSTEM

B. Niamir

This blank page was inserted to preserve pagination.

MIT/LCS/TR-192

ATTRIBUTE PARTITIONING IN A SELF-ADAPTIVE RELATIONAL DATABASE SYSTEM

Bahram Niamir

January 1978

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

LABORATORY FOR COMPUTER SCIENCE

CAMBRIDGE MASSACHUSETTS 02139

ABSTRACT

One technique that is sometimes employed to enhance the performance of a database management system is known as attribute partitioning. This is the process of dividing the attributes of a file into subfiles that are stored separately. By storing together those attributes that are frequently requested together by transactions, and by separating those that are not, attribute partitioning can reduce the number of pages that must be transferred from secondary storage to primary memory in order to process a transaction.

The goal of this work is to design mechanisms that can automatically select a near-optimal attribute partition of a file's attributes, based on the usage pattern of the file and on the characteristics of the data in the file. The approach taken to this problem is based on the use of a file design cost estimator and of heuristics to guide a search through the large space of possible partitions. The heuristics propose a small set of promising partitions to submit for detailed analysis. The estimator assigns a figure of merit to any proposed partition that reflects the cost that would be incurred in processing the transactions in the usage pattern if the file were partitioned in the proposed way. We have also conducted an extensive series of experiments with a variety of design heuristics; as a result, we have identified a heuristic that nearly always finds the optimal partition of a file.

The context of this study is a relational database management system that can process transactions made against relations whose physical partitioning is unknown to the user. In specifying and modelling this system, it is necessary to address the problem of optimizing nonprocedural queries made to a partitioned file. We have derived a number of such optimization techniques and have provided the results of a number of experiments with them.

ACKNOWLEDGEMENTS

I would like to acknowledge the contributions to this work of my thesis supervisor, Professor Michael Hammer; for his many invaluable ideas and criticisms; for his scrutinization and appraisal of the various drafts; for his continual encouragement and support; and for his willingness to put up with the perpetuity of this work.

I would like to express my thanks to Arvola Chan for many useful suggestions and discussions, and also to Dennis McLeod (both members of the Data Base Systems Group of the MIT Laboratory for Computer Science, formerly Project MAC) who have provided me with feedback on parts of the thesis.

Thanks are due to Chris Reeve, Tim Anderson, and other members and hackers of the Programming Technology Division of LCS for their help in the implementation effort and also for their assistance in the elimination, at times, of the weariness due to the preparation of this document.

I am indebted to those friends, who throughout these hard years, have never ceased with their moral support and companionship.

Finally, I would like to express my gratitude to my parents, Kazem and Suzie Niamir, who have provided me with unlimited amounts of patience, support, faith, and encouragement.

The work reported herein was supported by the Advanced Research Projects Agency of the Department of Defense and was monitored by the Office of Naval Research under contract numbers N00014-75-C-0661 and N00014-76-C-0944.

This report reproduces a thesis of the same title submitted to the Department of Electrical Engineering and Computer Science on January 15, 1978, in partial fulfillment of the requirements for the degree of Master of Science.

TABLE OF CONTENTS

Page

ABSTRACT 2

ACKNOWLEDGEMENTS 3

TABLE OF CONTENTS 4

Chapter

1. INTRODUCTION 7
   1. Self-adaptive Database Management Systems 7
   2. The Relational Model for Databases 9
   3. Attribute Partitioning in a Relational DBMS 12
   4. Thesis Objective 16
   5. Thesis Organization 17

2. THE APPROACH TO DATABASE ATTRIBUTE PARTITIONING 18
   1. Summary of Previous Work Done in Attribute Partitioning 18
   2. The Integer Programming Approach to Attribute Partitioning 25
   3. The Heuristic Approach to Attribute Partitioning 27
   4. Block Diagram of the Attribute Partitioning System 29

3. THE MODEL OF THE DATABASE MANAGEMENT SYSTEM 33
   1. The File Model 33
   2. Linking Subtuples 38
   3. The Index Organization 40
   4. The Transaction Model 41
   5. Query Processing in a Partitioned Database 45
   6. Parameter Acquisition 64

7. Repartitioning Points 71

4. COST ANALYSIS AND THE FILE COST ESTIMATOR 73
1. Sequential Search 75
2. Tuple Retrieval Using Links 75
3. Index Accessing and Tuple Retrieval 79
4. File Cost Estimation 80
5. Repartitioning Cost 83

5. THE ATTRIBUTE PARTITIONING HEURISTICS 85
1. The Exhaustive Enumeration Approach 86
2. The Stepwise Minimization Heuristic 87
3. The Pairwise Grouping Heuristic 89
4. File Cost Estimation as a Measure of Block Attractivity 97
5. The Fast Pairwise Grouping Heuristic 101
6. Other Variants of the Stepwise Minimization Heuristic 105
7. Experimenting with the pairwise grouping heuristics 114

6. AN EXAMPLE OF ATTRIBUTE PARTITIONING 132
1. The Statement and Solution of the Problem 133
2. The Efficacy of the Trivial Partition 139

7. REMARKS AND FUTURE DIRECTIONS 143
1. Extensions to the Attribute Partitioning System 143
2. Directions for Future Research 146

REFERENCES 153

CHAPTER 1

INTRODUCTION

The work to be reported in this report is part of an ongoing research effort to develop a self-adaptive database management system. The intent of this development is twofold: to develop the techniques and methodology for the construction of such systems, and to identify database physical design issues with techniques for their automatic and optimal determination. In this report, we address the problem of optimizing the performance of a self-adaptive database management system, in a dynamic environment where access requirements are continually changing, by automatically partitioning the attributes (fields) of the file. Attribute partitioning is the task of dividing the attributes (fields) of a file into non-overlapping groups and then storing each group in a separate physical file.

1. Self-adaptive Database Management Systems

It is important that a database system perform efficiently at all times. Efficient performance requires that the physical organization of the database match the usage pattern of its users. Thus, as the database's usage pattern changes over time, its organization and its access structures can become obsolete, with consequent degradation of performance. Performance degradation may also result as the database grows in size or as the nature of the data it contains changes. After some time, the performance of the database system may deteriorate sufficiently so as to compel a database reorganization. Since the applications

programs accessing the database are continually being altered with new applications programs replacing old ones, and since the contents of the database continually undergoes change, the reorganization of the database's physical structure must be an ongoing process.

Conventionally, the database administrator determines when and how to reorganize a database. His decision is based on limited information about the database and the transactions performed on it; consequently his decision is largely an intuitive guess. For large databases, a more systematic means of acquiring information about database usage and a more algorithmic way of evaluating the costs of alternative configurations is essential. It has recently been proposed that database management systems be self-adaptive, and automatically reorganize themselves as the need arises [35, 12]. Hammer [16] discusses a methodology for monitoring database usage pattern, and describes the design principles for a self-adaptive, self-reorganizing database management system.

A minimal capability of a self-adaptive database management system should be the incorporation of a monitoring mechanism that collects usage statistics while performing transaction processing. A database management system is well suited for gathering and analyzing information on its own usage and performance; and if the gathering and analysis of the usage and performance information is done appropriately, the associated overhead can be minimal. In addition, a self-adaptive database management system should be able to come up with desirable physical organizations (i.e., desirable data structures and access structures) based upon the collected statistics, and be able to evaluate the cost of each alternative organization in order to select an optimal physical organization for a database. Also, it is possible that such a system might perform the necessary database reorganization itself, after it has evaluated the cost/benefits of reorganization and the associated costs of retranslating the

applications programs that access the database.

2. The Relational Model for Databases

In a self-adaptive database management system, the physical organization of the database is perpetually being reorganized. In order for the database reorganization to be truly effective, a database management system that performs self-reorganization will have to manifest the following two important characteristics: 1- data independence between the database's physical organization and the application programs that access the database, and 2- nonprocedural access of the contents of the database. By data independence we mean that users and their application programs are not required to know the actual physical organization used to represent the data, so that they are free to concentrate on a logical view of the data. Data independence makes the database easy to use and avoids the need for application program retranslation every time the database's physical structure is changed. Nonprocedural access also makes the database easy to use; this entails the provision of access languages which allow the specification of desired data in terms of properties it possesses rather than in terms of the search algorithm used to locate and retrieve it.

The relational model of data (Codd [9]) has been proposed as a means of achieving the above goals of data independence and nonprocedural access. The relational data model provides a simple and uniform logical view of the data that is completely independent of the actual storage structures and access structure used to represent and access the data. This makes the definition and manipulation of a database independent of its underlying physical organization. As a result, changes at the data organization level need not be reflected in the

programs that access the database.

A relation in the relational data model is a named two dimensional table that has a fixed number of attributes (columns) and an arbitrary number of unordered tuples (rows). All the rows of the table have to be unique. A tuple representing an entry in a relation contains a value for each attribute of the relation. The number of attributes in a relation is m and the number of tuples in the relation is n. Figure 1 shows the relation ENROLLMENT for students enrolled in courses. The ENROLLMENT relation has two attributes *Student* and *Course*, and four tuples (Doe, 6.035), (Poe, 6.032), (Doe, 6.851), and (Roe, 6.035). The physical realization of a relation is often called a file, with the attributes and the tuples of the relation called the fields and records of the file respectively. Henceforth, we will use the term file when discussing the totality of the data in a relation, indicating that we are dealing with the physical representation of the relation. However, we will continue

ENROLLMENT:

Attribute

	Student	*Course*
Tuple	Doe	6.035
	Poe	6.032
	Doe	6.851
	Roe	6.035

Figure 1 The ENROLLMENT relation with 2 attributes and 4 tuples.

using the terms attribute and tuple for the fields and records of a file so that the two dimensional tabular data format of the relation will be kept in mind.

In a relational database management system, the user's view of the database is independent of the details of the database's physical organization. Furthermore, his nonprocedural queries are far removed from the primitive data manipulation operations for locating and retrieving the data. Consequently, more responsibility is placed on a relational database management system than on a conventional system. This responsibility takes two forms: 1- choosing an efficient physical organization for the relation, and 2- optimizing the process of finding answers to queries made to the database, by the means of efficient and judicial use of the available access structures.

We believe that the selection of a good physical organization is the primary issue in relational database implementation, since the efficiency that can be achieved by a query optimizer is strictly delimited by the available access structures. Furthermore, the efficient utilization of a database is highly dependent on the optimal matching of its physical organization to its access requirements, as well as to the other database characteristics (such as the distribution of attribute values in it). Hence, the usage pattern of a database should be ascertained and utilized in choosing the physical organization.

There are numerous possibilities for the physical organization of a relational database. The selection of a particular physical organization must be based on minimizing the performance cost in terms of both data access cost and data storage cost. The subject of this research is selecting the optimal attribute partition of a relational database by utilizing the access pattern history of the database in order to minimize the data access cost. Attribute partitioning consists of dividing the attributes of a file into subfiles that are stored separately.

In relational terms, this means splitting a relation into a number of subrelations, each containing a subset of the attributes of the original relation, such that the original relation may be uniquely reconstructed from the collection of the subrelations. (Strictly speaking, a subfile is not a relation in that duplicate tuples are allowed. We will give a formal definition of a subfile in the next section.)

3. Attribute Partitioning in a Relational DBMS

Let A be the set of attributes of a relation, and let T be the set of tuple identifiers of the relation. (A tuple identifier is a unique identifier for a tuple in the relation.) The number of attributes in the relation is $|A| = m$, and the number of tuples in the relation is $|T| = n$. Consider the collection of subfiles $F = \{F_i\}_{i=1}^{M}$, where each subfile F_i is defined by a pair consisting of an attribute set and a tuple identifier set, which specifies the attributes and tuple identifiers that are represented in the subfile $F_i : (A_i, T_i)$, $A_i \subseteq A$, $T_i \subseteq T$. The collection of subfiles F is called the clustering of the relation, and can have two basic forms:

1- an attribute cluster in which

$T_i = T$ $\quad i = 1, \ldots, M$ and

$\bigcup_{i=1}^{M} A_i = A$,

2- a tuple cluster in which

$A_i = A$ $\quad i = 1, \ldots, M$ and

$\bigcup_{i=1}^{M} T_i = T$.

The tuples of a subfile are called subtuples. A subfile F_i of an attribute cluster contains n subtuples, one for each tuple in the original relation. A subtuple of a subfile is

that part of the original file's tuple that has attributes A_i. The subtuples of subfile F_i in an attribute cluster need not all be different. For example, if the relation of Figure 1 is clustered such that $F_1 = (A_1, T)$ is a subfile with $A_1 = \{Course\}$, then the subtuples of F_1 will be (6.035), (6.032), (6.851), and (6.035).

An attribute cluster $\{F_i\}_{i=1}^{M}$ in which $A_i \cap A_j = \phi$ for all $i \neq j$ is termed an attribute partition of the relation. In this report, we will limit our attention to the topic of attribute partitioning. (A discussion of tuple partitioning appears in Section 7.2.) Attribute partitioning is the task of dividing the attributes of a relation and storing each disjoint subset of attributes in a separate subfile. The objective of attribute partitioning is to construct an attribute partition of a relation that optimizes the performance of the database management system by minimizing the cost of locating and retrieving data. Intuitively, attribute partitioning is accomplished by assigning attributes to the same subfile whenever they are consistently requested together by queries.

In conventional database management systems (with paged memory organization), each tuple of a relation is stored with all its attributes together in one file. When a query is made to the database, all tuples that are required by the query are brought into primary memory by retrieving all the pages that the tuples reside on. It has been observed in practical database applications that a query does not usually request all the attributes of the file; most queries request only a few of the attributes. Problems are presented by the co-existence in the same file (or equivalently in the same tuple) of attributes that are not requested by the query together with the few attributes that are. Whenever the requested attributes are retrieved, the non-requested attributes will also have to be brought into primary memory. If only a single tuple is needed to answer such a query, then it really does

not matter that other attributes that are not requested happen to reside in the same tuple with the requested attributes; in any event, only one page needs to be retrieved from secondary storage. On the other hand, usually more than one tuple must be retrieved in order to answer a query. This means that more than one page must be retrieved from secondary storage. The expected number of pages that must be retrieved for queries that require more than one tuple is inversely related to the number of tuples that fit onto one page. (The larger the number of tuples that can fit onto a page, the fewer pages that need to be accessed, since there is then a higher probability that two of the requested tuples will fall on the same page.) The effect of the non-requested attributes is to lengthen the tuple over what it must minimally be, thus reducing the number of tuples per page, and consequently causing excess page accesses when answering queries that request only a few of the attributes while accessing more than one tuple. Therefore if a file is partitioned such that attributes that are consistently requested together by queries are placed together into the same subfile and separated from those attributes with which they are not requested, then the number of page accesses required to retrieve these attributes will be reduced over the number required from a file that is not so partitioned.

On the other hand, indiscriminately separating all attributes and storing each in a separate subfile will also result in excess page accesses. This is because a query that requests the attributes of a subfile together with some other attributes that are not in the subfile will incur more page accesses than when all these attributes are in the same subfile, since now the two groups of attributes reside in different subfiles and on separate pages. When a file has been partitioned into subfiles, queries requesting attributes stored together in one subfile will become less costly to answer while those queries that have their requested attributes

distributed over more than one subfile will become costlier to answer. Intuitively, the optimal partition is the partition that maximizes the cost reduction for the first kind of queries while minimizing the cost increase for the second kind of queries.

Attribute partitioning is most useful for large databases where queries made to the database usually request only a few attributes of each tuple. It is conceivable that the request distribution that has been observed for tuples requested by queries also applies to attributes requested by queries. It has been observed in many practical database applications that not all the tuples of a database are requested with the same frequency. The "80-20" rule of thumb for the distribution of tuple request frequencies (Heising [17]) states that approximately 80 percent of queries request the 20 percent most frequently requested tuples in a file. Furthermore, the rule also applies to the 20 percent most frequently requested tuples in the file; i.e. that 64 percent of the queries request 4 percent of the most frequently requested tuples, and so forth. If this is also true for the request frequencies of attributes by queries, then most requests are only for a few active attributes of the file.

An example of a large database, where attribute partitioning may be useful, is the Navy Command and Control Data Base [9]. This database consists of a few relations with many tuples per relation. Some of these relations have as many as 35 attributes and a tuple length of 64 words. Queries are on-line, and predominantly involve only a few attributes. Some attributes of the file like the name of ships or the class of ships are frequently requested by queries while other attributes like the diameter of the torpedo tube are seldom requested. Therefore, partitioning the attributes of the files may result in considerable savings in page access requirements.

4. Thesis Objective

To summarize, the principal goal of this report is to develop techniques for attribute partitioning in a self-adaptive relational database environment. For this purpose, we have assumed a database management system that supports partitioned files and we have built an attribute partitioning system consisting of a model for the assumed database management system and a set of attribute partitioning heuristics. The attribute partitioning heuristics select a partition for a database managed by a database management system similar to the one we have modelled. Although our model is not one of any existing system, it is representative of practical systems. Our thrust in building this model has been to avoid many of the simplifying assumptions made in previous studies, and thereby emphasize important aspects of realistic database environments. We stress the need for monitoring the database management system and acquiring parameters on the database usage pattern and on the evolving characteristics of the database itself. We describe a methodology for processing transactions made to a partitioned database built with various access structures, and develop a complete and accurate model of the cost of accessing the subfiles when performing such a transaction. Finally, we concern ourself with heuristic techniques that utilize the acquired parameters and produce optimal or near optimal attribute partitions for the database at a reasonable computational cost.

5. Thesis Organization

The rest of this report is organized as follows. Chapter 2 summarizes a number of previous studies in the area of attribute partitioning, and in the context of evaluating them, argues for the need of a heuristic solution to realistic database attribute partitioning problems. In Chapter 3 we provide the model of the underlying database management system that we have considered: the physical storage structure, the access structures, the transaction model, the method of processing queries in the partitioned environment and techniques for the acquisition of the parameters needed for our cost analysis. In Chapter 4 we present the cost analysis for various basic operations on a partitioned database and describe how to compute the database's performance cost, which is what we wish to minimize. Then in Chapter 5 we present a number of attribute partitioning heuristics that we have devised, along with the motivation for their consideration. We also discuss the comparative advantages and disadvantages of each heuristic, and outline how each heuristic has performed in a series of experiments. Chapter 6 poses an attribute partitioning problem for a relation with 8 attributes, and describes its solution using the heuristics of the preceding chapter. Finally, Chapter 7 concludes the report with suggestions on how to extend the underlying environment in order to solve more realistic attribute partitioning problems, and also discusses the relationship between database attribute partitioning and other physical database design issues.

CHAPTER 2

THE APPROACH TO DATABASE ATTRIBUTE PARTITIONING

The purpose of this chapter is to introduce the approach we have taken to solving the attribute partitioning problem, and to contrast it with the approach taken by others in determining the optimal attribute partition. There are two major approaches to attribute partitioning, each approach having its own merits and limitations. The two approaches are: 1- the integer programming approach, which is the approach taken by most other researchers, and 2- the heuristic approach. We have chosen the heuristic approach for the following reasons: 1- More complex database environments can be handled by the heuristic approach than by the integer programming approach, 2- An optimal or near optimal attribute partition can be found much more efficiently by the heuristic approach than by the integer programming approach, and 3- Although the heuristic approach (unlike the integer programming approach) does not guarantee that the optimal partition will eventually be found, the heuristics we have employed have consistently found an optimal or near optimal partition for the attribute partitioning problem.

1. Summary of Previous Work Done in Attribute Partitioning

The idea of clustering attributes (and also attribute partitioning) as a means of improving the performance of a database management system has often appeared in the literature on file design and optimization. Until the paper of Kennedy [21], there had been

little systematic study of this aspect of file organization. Further, the conversion of a relation into second and third normal forms [10] was sometimes confused with attribute clustering. Although normalizing a relation into its normal forms may result in the clustering of attributes, and thereby reduce page accesses, normalization is directed towards improving the logical data schema rather than enhancing database system performance. It is the functional dependencies among the attributes that govern the splitting of relations in the process of normalization, rather than the data's physical characteristics or the database usage pattern. An example of work in the area of relational database normalization is that of Delobel and Casey [13]. They are concerned with the problem of decomposing relations into a set of subrelations such that the information content and logical relationships of the original relation schema are preserved. However, they do not consider physical database criteria that would result in a physically optimal decomposition of the relation schema.

Implementations of database management systems that support partitioned files have been few, and have been limited to simplified environments where finding an optimal or a suitable partition is relatively easy to manage. Moreover, in these implementations, attribute partitioning has been treated only as a one-shot affair, to be determined at the initial creation of a file. Attribute partitioning has not been viewed as a database organization issue that needs to be reconsidered periodically, where the retuning should be done by a self-adaptive database management system.

There have been a number of previous studies of attribute partitioning and attribute clustering (Day [11], Seppälä [32], Osman [29], Yue and Wong [39], Benner [4], Alsberg [1], Babad [2], Stocker and Dearnley [35, 12], Kennedy [21, 20], Eisner and Severance [14], March and Severance [23], and Hoffer and Severance [18, 19].) However, we feel that the results of

these studies are not directly applicable to a complete or realistic database environment. Some of these have been formal analyses which have made many simplifying assumptions in order to obtain analytic solutions; others have been designs that are incomplete or unrealistic in many ways. Our thrust here is to relax many of the simplifying assumptions that have been made in previous studies and thus to develop more complete and accurate models of cost and database access. In addition, we stress the importance of database cost analysis and the acquisition of accurate parameters, in an environment where access requirements are continually changing. This aspect of the attribute partitioning problem has not been addressed in previous work. Below, we present a summary of some of the earlier efforts in attribute partitioning, identifying the assumed environment of each project along with the thrust of its research.

Two of the earliest papers on attribute clustering in a self-adaptive database management system are by Stocker and Dearnley [35, 12]. They discuss the implementation of a self-reorganizing database management system that carries out attribute clustering. (Recall that in an attribute cluster, an attribute may exist redundantly in several subfiles.) Stocker and Dearnley show that in a database management system where storage cost is low compared to the cost of accessing the subfiles, it is beneficial to cluster the attributes, since the increase in storage cost will be more than offset by the saving in access cost. Although they do not outline the attribute clustering technique they use, they discuss a query processor, which utilizes graph theory and various heuristics to process queries made to a file clustered by its attributes. They conclude that attribute clustering in existing database management systems is both viable and desirable.

Kennedy [21, 20] considers a mathematical model of attribute partitioning where each

attribute a_i is of known length, and has probability p_i of being requested by a query. The joint probability that attributes a_i and a_j are requested by the same query is assumed to be $p_i p_j$; i.e., attributes are assumed to appear in queries independently of one another. A cost function based upon this assumption is derived, which reflects the expected amount of data that must be transmitted (in terms of number of words, from secondary storage to primary memory) in order to answer a query. The objective here is to choose a partition such that this cost function is minimized. Kennedy's model is a mathematical formulation of a simplified attribute partitioning problem in terms of zero-one integer programming where the only parameters are p_i and l_i, the length of attribute a_i. In addition to many other simplifications, Kennedy's model assumes that when an attribute is requested by a query, the subfile containing that attribute has to be retrieved and scanned in its entirety (rather than retrieving just those subtuples of the subfile that are really needed to answer the query). Since in this model, optimality can always be trivially attained when each subfile contains exactly one attribute, the number of subfiles M over which the attributes are to be distributed has to be fixed beforehand. (Otherwise the trivial partition, defined as the partition where each attribute is in a separate subfile of its own, will always prevail.) As Kennedy notes, there is no way short of exhaustive enumeration (which is infeasible as shown in Chapter 5) to find the optimal solution even for this rather simple model. To find the optimal solution to the partitioning problem posed by his model, he introduces two further simplifying assumptions in order to reduce the integer programming problem to a simpler problem where mathematical optimization techniques can be applied. One simplification is the assumption that all attribute request probabilities are equal, and the other simplification is that all attributes are of equal lengths.

In the work of Eisner and Severance [14], a file can be partitioned into two subfiles: a primary and a secondary subfile. Each subfile is located on a separate storage device characterized by differing storage cost and retrieval speed. The attributes are assigned to each of the subfiles without redundancy. Two forms of the objective cost function that is to be minimized are derived, where the first cost function is a special case of the (second) more general cost function. The first cost function is the sum of storage charges for subtuples in the primary subfile, plus the cost of accessing all the subtuples residing in the secondary subfile. (The secondary subfile is accessed only when a query requests an attribute which happens to be residing there.) This cost function is linear and may be solved by existing integer programming techniques. The cost of finding the optimal partition for this function by integer programming is of the order $(m + |Q|)^3$, where m is the number of attributes in the file and Q is the set of queries made to the database. The second objective cost function is nonlinear, and measures the total costs of access, transfer, and storage for subtuples in both the primary and secondary subfiles. The search cost for finding the optimal solution for the general nonlinear objective cost function is even higher than for the simplified linear cost function. The limitations of the model adopted by Eisner and Severance are apparent: only a maximum of two subfiles are allowed and the cost associated with processing a query is taken to be the cost of accessing the whole (primary or secondary) subfile in its entirety rather than the cost of retrieving just those subtuples of the subfile that are really needed to answer the query. Furthermore, the cost of finding the optimal partition using the linear objective cost function grows in the cube of the sum of the number of attributes and the number of queries (and the cost grows even faster for the nonlinear objective cost function); this cost is too large for practical purposes.

March and Severance [23] extend the model of Eisner and Severance to some extent by assuming that subtuples are blocked in each subfile into fixed size pages. (The page sizes in the primary and secondary subfiles are not necessarily the same, but the constraint is imposed that the sum of the primary subfile page size and the secondary subfile page size is constant.) The nonlinear objective cost function they derive not only depends on how the attributes are partitioned among the two subfiles, but also on the page sizes selected for each of the primary and secondary subfiles. Besides the rather peculiar paging organization adopted, the model of March and Severance has the additional disadvantage that it does not contain an accurate model of the cost of accessing subtuples that are selected in queries. Rather, the primary and secondary subfiles are assumed to be accessed in their entirety whenever any of their attributes are requested by a query (as in the model of Eisner and Severance). Using integer programming techniques, March and Severance obtain the optimal partition for their model. However, compared to the model of Eisner and Severance, the cost of solving the integer programming formulation grows even faster as the number of attributes and the number of queries made to the database grows.

Hoffer [18] developes an extensive model for attribute partitioning, in which the objective cost function is a linear combination of storage, retrieval, update, and insert/delete costs. The problem is formulated in terms of a nonlinear zero-one integer programming problem, and is solved by a branch and bound algorithm. In applying the optimization algorithm to the formulation, it became obvious that problems of even modest size were computationally intractable. In order to use this model to obtain solutions to realistic problems, it became necessary to reduce the size of the feasible solution space to a point where optimization becomes economically feasible. To this purpose, Hoffer and Severance [19]

propose an attribute partitioning method that produces an initial and crude, but nevertheless reasonable, partition of the attributes. This partition is then passed as a starting point to the branch and bound algorithm of Hoffer [18]. The initial partitioning method of Hoffer and Severance uses the cluster analysis algorithm of McCormick et al. [21], which is heuristic in nature, to group the attributes together into blocks. The clustering algorithm takes a set of objects and utilizes a measure of "similarity" for all pairs of the objects. It then rearranges the set of objects such that pairs of objects with large similarity measure fall adjacent or nearly adjacent to one another. Hence clusters (or blocks) of objects can be identified such that every pair of objects within the cluster carries a large measure of similarity, and every pair of objects across cluster boundaries carries a small measure of similarity. Hoffer and Severance provide attributes as objects to the clustering algorithm. They also develop a similarity measure for any pair of attributes (called the pairwise attribute access similarity measure), which expresses the degree to which the pair of attributes is used together in queries. The similarity measure of a pair of attributes is obtained as follows: A subfile consisting of only the two attributes is assumed. When answering a query that requests one or both of the attributes, the subtuples of the subfile need to be retrieved. However, not all of the information retrieved is used for answering the query: some of the subtuples may not satisfy one of the attributes, and hence the information contained for the other attribute in this subtuple is of no use. The similarity measure for the pair of attributes for this query is defined as the ratio of the amount of useful data transferred to the total amount of data transferred from such a subfile. The access similarity measure is derived by considering the set of queries, the frequency of each individual query, and the fraction of tuples satisfying each query.

The queries that Hoffer and Severance consider can contain only one attribute in their selection component. This assumption restricts the applicability of their techniques. Also, the only access path that they allow is sequential searching and therefore the subfile that contains the attribute of the selection component of the query is searched in its entirety (however, only tuples required for projection are selectively retrieved from the other subfiles). As with the model of Kennedy, the criteria by which a partition is selected for the file is the fraction of useful data transferred from secondary storage to primary memory. Since with such a criterion optimality can always be attained with the trivial partition, as a result, the number of subfiles in the partition found by the clustering algorithm has to be specified in advance, and the optimization techniques of Hoffer and Severance only look for the optimal partition in a subset of the space of all possible partitions. There are also problems associated with the clustering algorithm that they use. In Section 5.3, we describe some of these problems.

2. The Integer Programming Approach to Attribute Partitioning

The two approaches to attribute partitioning that have been taken are the integer programming approach and the heuristic approach. Most earlier work on attribute partitioning falls in the former category. There are two major problems associated with the formulation and solution of the attribute partitioning problem in the integer programming approach: 1- The applicability of this approach is limited because of the undue simplifying assumptions made on the problem environment in order to obtain an objective cost function that is amenable to optimization. In a realistic database environment where the file has

many attributes and many queries are made to the database and there are many access paths available by which to access the data, the number of variables and constraints to consider is so large that it effectively precludes an integer programming formulation of the attribute partitioning problem for the environment. 2- Even after many simplifications are assumed in the database environment, the attribute partitioning problem usually reduces to solving a zero-one nonlinear integer programming problem to which no available mathematical programming technique can be applied. As Kennedy notes [21], no technique has been found (short of enumeration) to solve the limited partitioning problem that is expressed only in terms of attribute request probabilities and attribute lengths. For cases where mathematical programming techniques are available for solving the integer programming formulation, applying them to even modestly sized problems is computationally infeasible.

The simplifying assumptions on the problem environment that have been made by previous studies fall to a large extent in two categories. One is a limitation on the complexity of the queries that are made to the database. Query predicates are either assumed to consist of single equality conditions, or else database usage is crudely described by a set of attribute access probabilities that indicate the probability of an attribute being requested by a query. Correlations between attribute occurrences in queries are ignored. The other simplification usually adopted concerns the computation of the cost of answering queries in terms of the amount of information that must be transferred. In this regard, the effect of blocking tuples (and subtuples) into pages has been completely ignored. The blocking of tuples into pages has the effect of increasing the amount of information transferred per tuple access. However, this increase is not linear, since accessing any number of tuples that reside on the same page will result in only one page access. If these blocking effects are ignored, then the

partitioning problem has a trivial solution. Kennedy (Lemma 4.1, [20]) has shown that when the amount of information transferred is the sole criterion of the cost function, the optimal attribute partition is the trivial attribute partition, as described in Chapter 1. The reason for this is that the total access cost is non-increasing as the attributes are dispersed into an increasing number of subfiles, even if the attributes are inappropriately partitioned. Hence in studies where blocking effects are ignored, in order that the trivial partition not prevail, the number of subfiles into which the attributes are to be partitioned has to be artificially limited and prespecified.

3. The Heuristic Approach to Attribute Partitioning

The approach to attribute partitioning that we have taken in our work is heuristic in nature. In the heuristic approach, an optimal or near optimal partition is found for the attributes by a process of stepwise minimization. An attribute partitioning heuristic which is based upon stepwise minimization starts with a given partition (e.g. the trivial partition), and attempts to derive from it a new partition that is incrementally superior to the original one, in that the database partitioned according to the new partition will have a lower performance cost. When this is achieved, the heuristic further tries to improve upon the newly derived partition. Each time an improved partition is derived, the performance cost of the database is reduced. The stepwise minimization process is continued until no improvement can be made to the latest partition. This last partition will then be returned as the result of the attribute partitioning heuristic. The resultant partition is not necessarily optimal, although it can often be argued that the partition is near optimal. (The near

optimality of the partition proposed by the heuristic can be verified by comparing the performance of the database management system when the file is optimally partitioned with the performance when the file is partitioned as suggested by the heuristic.) Indeed, in the course of our experimentation with our attribute partitioning heuristics, we have consistently found that the resultant partition of the heuristics is either optimal, or differs only insignificantly from the optimal partition.

The heuristic approach to attribute partitioning does not suffer from the two major problems associated with the integer programming approach. The model of the the database environment may be as complex as desired. The complexity of the model adopted does not seriously hamper the heuristic's ability to find reasonable solutions to the partitioning problem (although it may affect the precise amount of search time required by the heuristic to find a reasonable solution). We note that, although our model does not consider certain parameters that have been considered by some earlier studies (e.g. file storage cost, overhead cost for accessing subfiles, different access and transfer costs for each subfile, and the imposition of constraints on the allocation of attributes to subfiles), we could readily incorporate these parameters into our model of the database management system and take them into consideration without needing to significantly alter our partitioning heuristics. (These extensions are described in Chapter 7.) The heuristic approach is also relatively more efficient with respect to the time needed to determine a solution. For example, the main attribute partitioning heuristic that we develop in Chapter 5 operates in time that is on the order of the product of the number of queries in the usage pattern and the square of the number of attributes in the file. This compares very favorably with execution time of the integer programming approach.

The model of the database management system we have adopted in this work is in many ways a generalization of earlier work, and although not a model of any particular existing system, is more reflective of practical systems than earlier models. We have allowed more complicated forms of queries, and have also considered the effect of blocking subtuples into pages. We allow a diverse set of access structures in our model, including links, indices, and segments. The objective cost function that we seek to minimize is the total cost of answering the queries posed to the partitioned database, and is expressed in terms of the number of page accesses, rather than in terms of the amount of data transferred. Unlike the models of previous studies, two tuples which happen to reside on the same page, if retrieved, will not incur twice the access cost of retrieving one of them. Conversely, a single tuple that has its attributes partitioned, if retrieved in its entirety, will cause numerous page accesses. Consequently, if the attributes of a file are partitioned inappropriately such that attributes that are requested together are placed in separate subfiles, the performance cost of the partitioned database increases. This contrasts with previous models where access cost was determined solely in terms of total information transferred (and so for which the trivial partition is always optimal). In our model, we do not need to specify M, the number of subfiles in the chosen partition. Rather, M is unconstrained and is determined by the heuristics according to the optimal partition.

4. Block Diagram of the Attribute Partitioning System

Our attribute partitioning system consists of four components. Figure 1 shows the block diagram of the system, in which each component appears as a box. The four

components are: 1- the parameter acquisitor and forecastor (described in Chapter 3), 2- the file cost estimator (described in Chapter 4), 3- the query processor (described in Chapter 3), and 4- the attribute partitioning heuristics (described in Chapters 5 and 6). The circles in the figure represent the collection of forecasted parameters, prepared by the parameter acquisitor and forecastor, characterizing the database and its usage. Edges in the figure are labelled by the kind of object passed from one component to another. A brief description of each component follows.

1- The parameter acquisitor and forecastor continuously monitors the usage pattern and the response of the database management system to the queries in the usage pattern. It collects the statistics needed as parameters by the file cost estimator and the query processor. At certain points in time when file repartitioning is initiated, the parameter acquisitor calculates trends and makes forecasts of the database usage pattern and database parameters for a time interval into the future.

2- The file cost estimator receives a proposed partition from the partitioning heuristics and evaluates it by finding the cost of processing each query in the forecasted usage pattern against the accordingly partitioned file. To compute the cost of processing a query, the file cost estimator passes the query to the query processor for query analysis. The query processor finds a method for the query and returns the method to the file cost estimator. A method for a query is a procedure indicating how to go about accessing the subfiles in order to answer that query. Using the forecasted database parameters for the future time interval, the file cost estimator computes the number of page accesses required to answer the query against the partitioned file according to the query's method. Summing

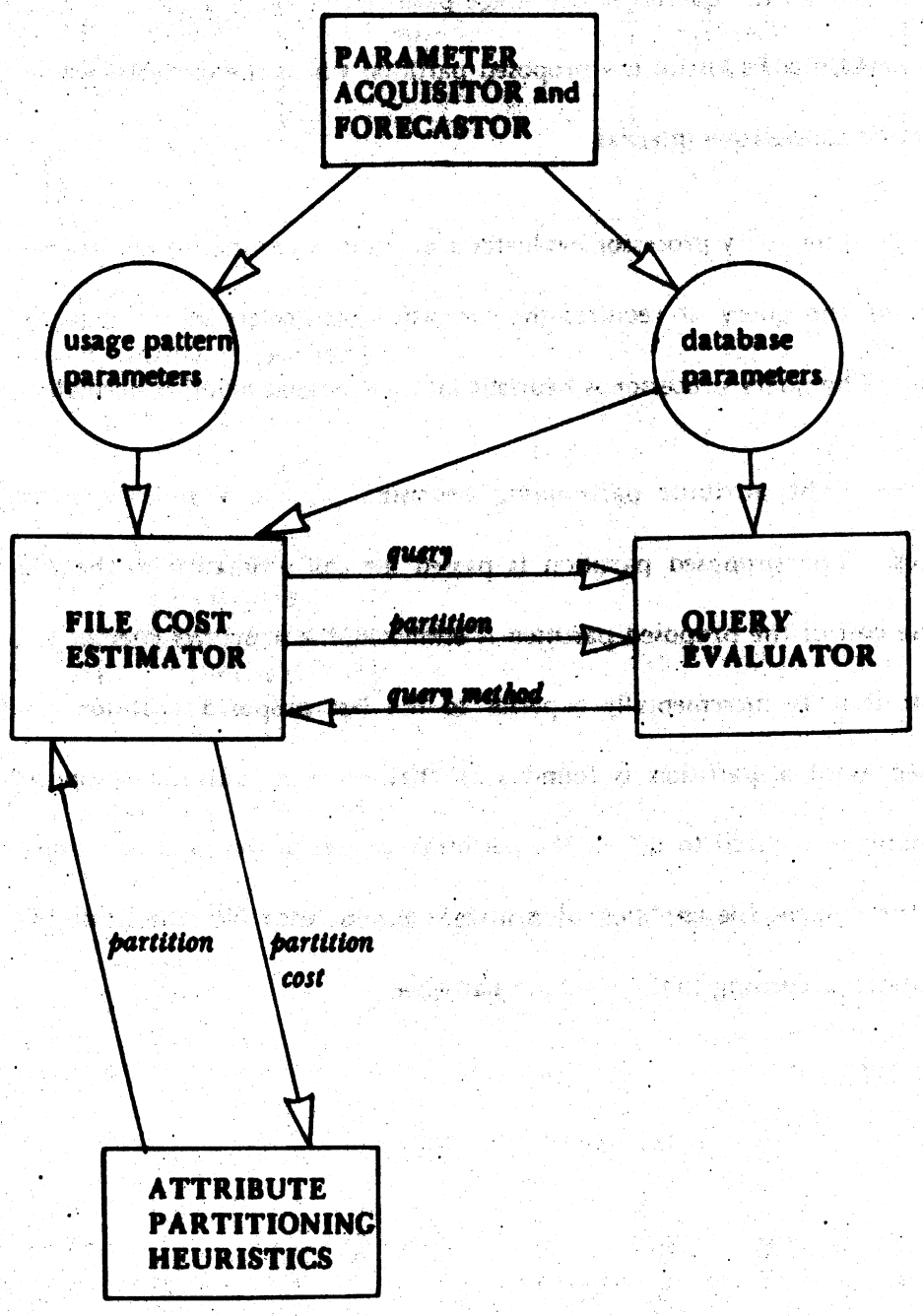

Figure 1 Block diagram of the attribute partitioning system.

these costs for all the queries in the usage pattern, the file cost estimator obtains an estimate for the performance cost of the proposed partition which the partition would be expected to incur in the future time interval.

3- The query processor evaluates a query in a partitioned environment by finding a method for the query. It requires the forecasted parameters of the database and the file partition. The query processor is heuristic and the method found is normally near optimal.

4- The attribute partitioning heuristics propose a suitable partition of a file's attributes. The proposed partition is passed for cost estimation to the file cost estimator. After the cost of the proposed partition is estimated, the heuristics attempt to come up with a partition that is incrementally superior to the last proposed partition. This process is continued until a partition is found such that no other partition proposed has a better performance compared to it. If the performance cost of the final partition is less than the cost of the current file partition by a margin that exceeds file repartitioning cost, the file is repartitioned according to the resulting partition.

CHAPTER 3

THE MODEL OF THE DATABASE MANAGEMENT SYSTEM

In this chapter we will describe the underlying model of the database management system that we have assumed in our work. We will describe the storage structure and the access structures we have adopted for the physical representation of a relation, and the assumptions we have made for the purpose of reducing the problem of attribute partitioning to a manageable size. We will then describe the structure of the queries made to the database and the strategy employed to process the queries in a partitioned environment. Finally, we will list the parameters required by the components of our attribute partitioning system (the attribute partitioning heuristics, the file cost estimator, and the query evaluator), and describe how these parameters are obtained from monitoring the operations of the underlying database management system.

1. The File Model

We have chosen the relational data model as the logical view of data for a database. A database in the relational context consists of one or more relations. However, in order to make the problem of attribute partitioning manageable in size, we address the reduced problem environment of a database with a single relation. In addition we assume that the physical implementation of a relation is a flat file. That is, a relation is stored as a set of unordered contiguous tuples in secondary memory. There are no hierarchies of domains, nor

pointers from one tuple to another. Although the assumption of flat file storage structure may seem rather severe, we note that this is the most natural way of storing a relation. Also, some of the drawbacks of the flat file storage structure, such as the placement of frequently used data together with seldom used data in the same physical locality, is precisely what attribute partitioning intends to eliminate. We note here that although the work reported here is based on the assumption of a single relation database and a flat file storage structure, the approach to attribute partitioning that we have taken and the attribute partitioning heuristics that we have developed should be extendible to problems where any of the two assumptions are relaxed. Specifically, if there is a facility available to estimate the cost of answering a query made to a multi-relational database with a non-flat file storage structure, then the main heuristic techniques that we have developed may be regarded as a viable candidate for the purpose of attribute partitioning. For further discussion of the possible relaxation of the above two assumptions, refer to Section 7.2.

All the subfiles of the attribute partition are assumed to reside on direct access secondary storage devices like disks [6]. Storage space on such devices is divided into fixed size blocks called pages. A page is the information quantum transferred between the disk and primary memory in one disk access. The accessing cost of a page is assumed to be proportional to the average disk seek and latency times plus the page transfer time. Hence, accessing cost will be independent of the sequence of page accesses. Consequently, we may think of the pages of a file scattered throughout the disk, with no restriction on their relative physical locations.

As mentioned above, the tuples of a relation are stored unordered with respect to any attribute. The order in which the tuples are stored will be their chronological order of

insertion into the file. This makes the problem of file maintenance due to updates, insertions, and deletions much simpler. If a tuple is updated, the new values replace the old values in the same tuple. A tuple that is deleted is joined to a pool of deleted tuples and will be reused for newly inserted tuples. (Such a pool can easily be maintained by threading the tuples that were deleted into a list.) A new tuple that is inserted in the file replaces a tuple that has been previously deleted. If the pool of deleted tuples is empty, then the inserted tuple is appended to the end of the file (if the file occupies an integral number of pages, a new page is allocated to the file).

The above strategy for overflow handling is intended to maximize the number of undeleted tuples per page, and keep the file size to a minimum. The cost of a sequential search and tuple retrieval by the link access path (described below) are inversely related to the (average) blocking factor (the average number of used tuples per page), and these costs should be minimized by keeping storage utilization in the tuple space as high as possible. Even with the above assumptions, poor storage utilization may still ensue if the database usage pattern consists of a large number of insertions followed by an equally large number of deletions. To correct this, garbage collection may be performed on the tuple space so the tuples are recompacted to occupy as little space as possible. We note here that in partitioning the attributes of a file, the cost of garbage collection may be eliminated from consideration and that if we ignore the effect garbage collection has on the subfile blocking factor, the optimal attribute partition is independent of the garbage collection cost. The reason for this is that no matter how the file is partitioned, garbage collection of deleted tuples requires that the entire file be brought into primary memory and shipped out back onto secondary storage. Since the total amount of storage is fixed regardless of how the file is

partitioned (except for page breakage at the end of each subfile, which is negligible), the cost of garbage collection does not enter the optimization process. On the other hand, the blocking factors of the subfiles do influence the optimal partition. The more frequently that the file is garbage collected, the fewer the number of unused tuples per page and the larger the blocking factor would be on the average for the file. Therefore, the optimal partition for the file will partially depend on the frequency of garbage collection. Since the optimal selection of points where the file is to be garbage collected is in itself another database design optimization problem, we will not consider the problem of the optimal determination of garbage collection points. (See the works of Shneiderman [33] and Yao et al. [38] for a discussion of this problem.) We will assume that the subfile blocking factor that the parameter acquisitor prepares for the attribute partitioning heuristics is the overall average of the observed blocking factors throughout the planning horizon.

We will assume that tuples are of fixed length (i.e. each tuple occupies the same amount of storage space), so that each page has a capacity for a fixed number of tuples. This implies that attribute values have fixed sizes, since a normalized relation has a fixed number of attributes per tuple. This assumption is in correspondence with the relation being a flat file and is a necessity for the realization of links between subtuples of the same tuple. We also make the assumption that each page contains an integral number of tuples, and that tuples do not overlap page boundaries.

In our file model, we allow three kinds of access structures. These are: segments, links, and indices. An access structure is a mechanism that makes the search and retrieval of tuples possible. In other words, given the value of an attribute, an access structure can locate and retrieve all tuples having that value for the attribute. The access path of an access

structure is the way in which the structure is used in such a search. A segment is a file or a subfile that may be retrieved into primary memory and sequentially searched from top to bottom for tuples with a certain attribute value. Hence by using the sequential search access path of the segment access structure, we can both locate desired tuples and retrieve these desired tuples at the same time. A link is an access structure for retrieving tuples which have already been located. In other words, assume that we have a pointer or some other identifier that uniquely identifies a tuple by its location. Linking is the access path for deriving the physical address of the tuple from the identifier and retrieving the tuple from secondary storage. Therefore the link access structure is a mechanism for retrieving tuples that have already been identified and whose location is known. A link cannot be used to search for tuples that possess a certain value (content retrieving). In our file model, each subtuple of a subfile has a link to all its corresponding subtuples in all the other subfiles. The corresponding subtuples (or co-subtuples) of a subtuple are all the subtuples that made up a single tuple before the file (and the tuple) was partitioned. An index is an access structure for locating subtuples with attributes that match certain values, without actually retrieving the subtuples. An index does not have the capability of retrieving tuples. In order to retrieve the tuples that have been located by an index, a link access structure is used. In our file model, any attribute of the relation may have an index; which ones are actually to be indexed is a separate database design issue.

2. Linking Subtuples

Sequentially searching a file (or a subfile) is a straightforward matter and we will not discuss it any further. A link, on the other hand, is an access structure that is widely used in our model and we will describe how linking is performed in detail.

Once a subtuple has been located or retrieved, it must be possible to retrieve any of its co-subtuples in the other subfiles. (A co-subtuple may have to be retrieved in order to see if an attribute of it contains a certain value, or in order to project one of its attributes.) Hence, we assume that each subtuple has a link to all its co-subtuples and that the co-subtuples may be retrieved by linking. Links as a means of relating tuples may be classified according to purpose, realization, and cardinality. The purpose of a link in our model is to relate co-subtuples. Its realization is logical, i.e. the link is derived by transforming the address of one co-subtuple into the address of another one. The cardinality of the link is one-to-one, i.e. each subtuple is linked to exactly one subtuple in another subfile. Thus, there is no explicit pointer from a subtuple of a subfile to each of its co-subtuples. Rather, the address of the co-subtuples in different subfiles can be calculated from one another's addresses. By subtuple or co-subtuple address we mean the tuple identifier (or the logical address of the tuple), which is the address of the tuple relative to the base address of its file. When retrieving a subtuple by using a link, the subtuple's tuple identifier (TID) is translated by the file's page map table into the physical address of the page the subtuple resides on and the offset of the subtuple within the page. The page address is then used to retrieve the subtuple from secondary storage in one page access. Note that when retrieving any number of subtuples that reside on the same page, the page needs to be retrieved only

once. Once we have the TID of a subtuple in a subfile (this happens when the subtuple has been located or retrieved), we may obtain the TIDs of all co-subtuples in other subfiles by applying a transformation to the subtuple's TID. This has been made possible because when partitioning a flat file implementation of a relation, all co-subtuples retain their relative position in their subfiles, and also because within a subfile all subtuples are of the same size. To see what this transformation is, let τ_i be the tuple with tuple number i (i.e. τ_i is the ith tuple of the file). If the file is partitioned into M subfiles, then $\tau_{i1}, \tau_{i2}, \ldots, \tau_{iM}$ are the M co-subtuples. Let t_{ij} be the TID of subtuple τ_{ij}, $j = 1, \ldots, M$. We want to calculate t_{ik}, the TID of τ_{ik}, from t_{ij} the TID of τ_{ij}. We first show how to get the tuple number i from the TID t_{ij}. Let S be the system page size, l_j the subtuple length in subfile F_j, and let $b_j = \lfloor S/l_j \rfloor$ be the number of subtuples per page in F_j (we have assumed that tuples or subtuples do not cross page boundaries); then $\lfloor t_{ij}/S \rfloor$ is the page number of τ_{ij}, and $(t_{ij} \bmod S)$ is the offset of τ_{ij} in its page. The tuple number i is therefore:

(3.2.1) $$i = b_j \lfloor t_{ij}/S \rfloor + (t_{ij} \bmod S)/l_j$$

Finally, we want to calculate t_{ik}, the TID of τ_{ik}, from i, its tuple number. Since $\lfloor i/b_k \rfloor$ is the page number for the subtuple τ_{ik} in F_k, and $(i \bmod b_k)$ is the number of the subtuple within the page, we have:

(3.2.2) $$t_{ik} = S \lfloor i/b_k \rfloor + l_k (i \bmod b_k)$$

3. The Index Organization

Another access structure we have considered in our model is an index. A file may have an index on one or more of its attributes. A subfile that contains an indexed attribute, has indexing as an access path to locate subtuples having a specified value for that attribute. In our model, we have chosen an index to be a balanced tree, where each node of the tree is a page. A detailed description of the index's structure may be found in Chan [7] and Blasgen and Eswaran [6]. The index is very similar to the B-trees of Bayer and McCreight [3] both in terms of structure and the way it is maintained. Briefly, each non-leaf page of the index contains an ordered set of pairs of keys (attribute values) and pointers, each pointer pointing to nodes in the next lower level of the tree. The key in the pair is the highest key of the node the pair points to. A leaf page consists of a key followed by an ordered list of TIDs of subtuples that have the key as the value of the indexed attribute in the subfile. The choice of index structure for our work is obviously not limited to balanced trees. Any index that lends itself to usage cost analysis and which is independent of the choice of file partition may be alternatively used.

To concentrate on the problem of attribute partitioning, we assume that the choice of indices and their structure is predetermined and chosen beforehand on the basis of other criteria besides the file partition. This is not to suggest that the problems of index selection and attribute partitioning are independent of one another. Indeed, the two problems are mutually interdependent and a better solution to the attribute partitioning problem can be achieved by their simultaneous solution. The problem of selecting indices that befit a database usage pattern has been extensively analyzed by Chan [7]. Our work on attribute

partitioning takes up another dimension of the general problem of physical database design.

4. The Transaction Model

We will consider four types of transactions that may be conducted against the database: queries, updates, insertions, and deletions. The query and the update transactions consist of two components: a selection component that determines the tuples that are to be selected, and a projection component that determines which attributes of the selected tuples are to be extracted and returned (in the case of a query) or updated (in the case of an update). The deletion transaction consists only of a selection component that determines which tuples have to be deleted. The insertion transaction has no components. An insertion transaction is basically a set of tuples that have to be inserted in the file. Because of the similarities among query, update, and deletion transactions, henceforth, we will discuss only one of them, namely queries, in full detail. The reader should assume that the discussion for queries can be generalized for the other transaction types as well. The only difference among the transaction types is how the projection component of each transaction type is processed after the tuples are selected. This difference in processesing the projection component will be delineated later.

We have made certain simplifying assumptions on the structure of the queries considered in our model. The simplifications were necessitated by the need to reduce the task of query cost analysis to a manageable size. We have disallowed join operations on the relation in queries. The boolean expression in the selection component of a query consists of either a conjunction made of equality conditions, or a disjunction made of equality conditions.

A query with just one equality condition is considered to be a special case of a conjunctive query. An equality condition is a predicate of the form (a = x), where a is an attribute name, and the attribute value x of the equality condition is a constant or program variable which is known at the time the query is processed. The equality condition in the selection component is used to search for all the tuples (subtuples) in the file (subfile) that have attribute value x for attribute a. The projection component is a set of attributes whose values are extracted from all tuples that satisfy the selection component and returned as the answer to the query. In a conjunctive query, an attribute cannot appear twice in the selection component, or appear both in the selection and projection components. Although we have restricted the set of allowable queries by the assumptions presented above, we have still included a large number of possible queries, encompassing many of the more frequent queries encountered in practical database applications.

When a query is made to a database, the query processor does the necessary search and retrievals on the database and returns the answer to the query. There is a cost associated with processing a query. In our attribute partitioning system, we have incorporated a query evaluator and a file cost estimator that can analyze a given query and provide an estimate of the cost of answering the query. Query cost analysis is a complex task. The assumptions we have made on the structure of the query alleviate some of the difficulties in query processing and query cost analysis. Besides the assumptions on the structure of a query, query cost analysis depends on the assumptions made on the distribution of attribute values in the file. Query cost analysis also depends on the distribution of attribute occurences in the selection and projection components of queries and the distribution of attribute values in the equality condition predicates of queries. As we have mentioned in Chapter 2, previous work done on

attribute partitioning made simplifying assumptions on the distribution of attribute values and on the distribution of attribute requests in order to keep the problem of query cost analysis (and hence the attribute partitioning problem) within manageable limits. We have also made simplifying assumptions on the distribution of attributes values and attribute requests in building our model of the database management system. However, our simplifying assumptions are less restrictive in nature than those made in the works of our predecessors and are closer to the realities of practical database usage. We have made the following two assumptions in our transaction model.

1- We assume the fraction of tuples that satify a one predicate selection is the selectivity of the attribute in the equality condition. The (average) selectivity of an attribute of a relation is the average fraction of tuples under consideration that have historically satisfied an equality condition involving that attribute. In other words, the selectivity of an attribute is the fraction of tuples that will most probably satisfy an equality condition on the file. The concept of an attribute selectivity measure is an important tool for database modelling and query cost analysis. The attribute selectivity measure will be defined and described fully in the section on Parameter Acquisition. From the attribute selectivities, the number of tuples that satisfy an equality condition on an attribute is estimated as the product of the selectivity and the number of tuples in the file. Using this measure of selectivity avoids the naive assumption that the attribute values are uniformly distributed in number, and that the number of tuples satisfying an equality condition is the total number of tuples divided by the number of different values of the attribute. Also by using this measure, we have avoided the simplistic assumption that attribute values of a given attribute occur with

equal probability in the selection components of queries. Although we could have obtained a still better model of value distribution by noting the number of tuples that contain each value of an attribute in a table, the attribute selectivity measure has the definite advantage that it takes little storage for its preservation. The other scheme requires that a table of attribute value frequencies be maintained for each attribute in the file, and if there are many distinct values for an attribute, this table will consume a significant amount of storage and will also be very difficult to update.

2- Since we allow the specification of queries with multiple equality condition predicates, it is necessary to have a measure for the joint resolving power of two or more equality conditions. (This measure is called the joint selectivity measure.) For this purpose, we will assume that the appearance in tuples of values belonging to different attributes is independent. (I.e., the probability that value x of attribute a and value y of attribute b appear in the same tuple is equal to the product of their individual probabilities of appearance.) Hence the fraction of tuples satisfying a conjunction of predicates simultaneously is the product of the fractions that satisfy each predicate, and the fraction of tuples satisfying a disjunction of predicates is the complement of the fraction not satisfying any of the predicates of the disjunction.

One assumption we do not make in our model, however, is that attributes occur independently of one another in the selection and projection components of the query. Neither do we make the less narrow but nevertheless still restrictive assumption that the correlation between attribute occurence in queries is determined by joint probabilities of attribute occurence. We actually keep a record (in a table, called the table of query types) of

all queries made to the database, and the exact correlation in the occurence of attributes in queries may be obtained form this table. Thus we avoid making the strong (and often inaccurate) assumption that an attribute is requested by a query independent of what other attributes are requested by that query. This table of queries is concise in that queries involving the same attributes but different attribute values are clustered together in one entry. (The number of queries in the cluster is also recorded in the entry.)

5. Query Processing in a Partitioned Database

An integral part of a database management system is a facility to decide how to answer queries. Since we are modelling a database management system that decides how to answer queries posed to the database, and since in the course of attribute partitioning we need to estimate the cost incurred in answering a query posed to the model database, our attribute partitioning system will also need to decide how to answer queries. When a query is made to the database, appropriate access paths must be chosen so that tuples satisfying the selection component of the query may be located. After the satisfying tuples are located (i.e. a TID list of such tuples is obtained), the same access path (or possibly some other access path) will have to be used in order to retrieve the tuples.

For example, assume we have a conjunctive query involving attributes $a_1, \ldots\ldots, a_L$ in the selection component and attributes $a_{L+1}, \ldots\ldots, a_K$ in the projection component made to a partitioned file. In order to answer the query, the subtuples that satisfy the equality conditions on $a_1, \ldots\ldots, a_L$ need to be located (by creating a TID list pointing to the subtuples), and then their co-subtuples containing attributes $a_{L+1}, \ldots\ldots, a_K$ have to be retrieved so that the

value of projection attributes a_{L+1}, ..., a_K may be extracted and returned. Assume that there are indices available on some of the attributes a_1, ..., a_L. We may proceed to locate the subtuples that satisfy the selection component in either of the two following ways (in the rest of this section, we do not explicitly specify when the transformations 3.2.1-3.2.2 are to be performed on TIDs of subtuples to get the TIDs of co-subtuples; we assume the transformations are performed whenever necessary):

1- Use all the applicable indices to retrieve the TID lists of subtuples satisfying the indexed attributes, intersect these TID lists (because the query is conjunctive), and from the resulting TID list link to the subfiles that contain any of the unindexed attributes a_1, ..., a_L (an applicable index is an index on a selection attribute). Subfiles are accessed one at a time. Everytime a subfile is accessed, its subtuples with TIDs in the list are retrieved (via links) and checked to see if they satisfy the equality conditions on the unindexed attributes. The TIDs of subtuples that do not satisfy any of the unindexed attributes are then pruned from the TID list (i.e., the TID list of the subtuples that satisfy all of the unindexed selection attributes in the subfile is intersected with the old TID list). After all the subfiles containing selection attributes have been accessed, and the TIDs in the list have been tested to satisfy the equality conditions, then all subtuples with TIDs in the list are retrieved (again by linking) from all the subfiles containing projection attributes, and the projection attributes are extracted.

2- Use none of the indices. Sequentially search one of the subfiles containing selection attributes, and create a TID list of the subtuples that satisfy all predicates involving the subfile. Thereafter, using the TID list, link one by one to the subfiles containing the remaining selection attributes, until a TID list of subtuples satisfying the entire selection

component of the query is obtained. Finally, link to subfiles containing projection attributes.

Each of the above two schemes may be thought of as a step by step procedure where at each step an access path (sequential searching, indexing, linking) is performed in order to obtain the TIDs of subtuples that satisfy one or more of the equality conditions in the selection component. Let us call the act of obtaining TIDs of subtuples that satisfy the equality condition on an attribute the act of resolving the attribute. Hence each of the above two schemes is a step by step procedure where at each step an index is used to resolve an attribute, or a sequential search/link is used to resolve one or more attributes in one subfile. We define the method of a query to be such a step by step procedure where at each step an access path is used in order to resolve one or more attributes.

A query usually has many different methods. For example, in the two schemes above, we chose either to use all the indices, or to use none. We might have chosen to resolve some of the indexed attributes (in the selection component of the query) by indexing, while resolving the rest of the indexed and unindexed selection attributes by linking. Similarly, when linking to subfiles, the subfiles will be accessed in some sequential order (i.e. one subfile is linked first, another subfile second, etc.) Each distinct subset of applicable indices and each distinct subfile sequence constitute a method of the query. (Hence each of the two schemes above may be translated into many methods as the sequence of linking to subfiles is instantiated.)

There is a cost associated with a query's method. Depending on what indices are used and in what sequential order the subfiles are linked, the cost of answering the query will be different. For example assume that in resolving the attributes of a query, two subfiles have to be linked and when each subfile is linked, the size of the TID list will be reduced by

an equal factor. Then it is better to link to the subfile with the smaller number of pages before we link to the subfile with the larger number of pages. Although in the first method the second link will result in more page accesses than the second link in the first method, the first link in the second method will result in even more page accesses than the first link in the first method. Therefore, it is important that a query processor consider all the methods of a query and select the method which results in the smallest number of page accesses when answering the query. The optimal method of a query will depend on the attributes in the selection component of the query, the attributes in the projection component, the attribute selectivities and lengths, the attribute partition, and on other database parameters. A query processor will have to consider all these parameters when choosing a method for a query.

The purpose of this section is to present how our attribute partitioning system goes about choosing a method for the query made to the partitioned database. Before we present our strategy for choosing methods, we delineate and describe the different phases of query processing; the first phase is the phase in which the query processor decides on the optimal method of the query.

Processing a query made against a partitioned database with a single relation and in which no joins or aggregate operators are involved consists of three phases: 1- query evaluation, 2- query resolution, and 3- query answering.

1- Query evaluation - Query evaluation is the process of finding the optimal method for a query. In an environment where the file is partitioned and attributes are indexed this means: 1- selecting the indices to use in answering the query, which could be selecting all, none, or some of the applicable indices, 2- selecting the sequence of accessing those subfiles

that contain selection attributes not resolved by means of indices. (Note that if no index is utilized by the method, then the first subfile of the method will be sequentially searched, while the remaining subfiles will be linked.) The agent for finding the optimal method for a query (or finding a suitable method in case the optimal method is difficult to find) is the query evaluator. The query evaluator chooses a method with the objective of minimizing page accesses when answering the query. Later in this section, we will present the strategy used by the query evaluator of our attribute partitioning system. The method chosen by our query evaluator is not necessarily the optimal method for the query, although we will show that our strategy results in near-optimal methods. When a satisfactory method is found for a query, we say that the query is evaluated.

Note that in our model of query evaluation, the query evaluator does not take into consideration the projection attributes of the query. Strictly speaking, the query evaluator should also take the projection attributes into consideration and the method of the query should specify the sequence of linking to the subfiles containing projection attributes. This is because the cost of answering a query is influenced by the sequence in which projections are made. For example, if a subfile contains both selection attributes and projection attributes, then it is beneficial that this subfile be linked last in the method; since if the subfile is linked last, both the selection attributes may be resolved and the projection attributes may be projected concurrently. If this subfile is not linked last, it will be linked once for resolving the selection attributes and another time for projecting the projection attributes. (Note that each time the subfile will be linked from a different TID list.) We have eliminated projection attributes from consideration when evaluating a query. We do this because: 1- considering projection attributes will make the problem of query evaluation still more difficult, and 2-

because we believe that the sequence in which the subfiles are accessed in resolving attributes has a more profound influence on the cost of answering a query than the sequence in which the subfiles are linked for projecting attributes.

The query evaluation performed by our query evaluator does not entail any input/output operations (i.e. page accesses). The query evaluator does not need to know about the actual data contents of the subfiles; it only requires the various parameters prepared by the parameter acquisitor. The query evaluator evaluates a query by choosing a method for it, utilizing some strategy. One such strategy is exhaustively enumerating all possible methods for the query, estimating the cost of answering the query according to each method (by using the file cost estimator), and then choosing the optimal method. We have discarded this strategy because it is computationally intractable to consider all possible methods for a query. This is especially true when making a query against a database partitioned into many subfiles and/or if there are many indices available. The strategy we use for query evaluation is instead based upon choosing a near-optimal method without requiring extensive analysis of the query.

Query evaluation is the only phase of query processing that is an optimization process. The other two phases of query processing do not attempt to optimize the cost of processing a query. Query evaluation is the only phase actually performed in our attribute partitioning system. The next two phases are only performed by a database management system when it actually processes a query. The reason our attribute partitioning system evaluates queries is that the method of a query is required in order to estimate the cost of answering the query. The query evaluator our system supplies the method of the query to the file cost estimator, which computes the cost of locating the selected subtuples (according to the

method) and the cost of retrieving the subtuples needed for projection. (The attribute partitioning heuristics require the cost of answering all the queries in the database usage pattern. See Section 4.4 for a detailed discussion on how the file cost estimator estimates the cost of answering a query.)

2- Query resolution is the process of locating the set of tuples that satisfy the selection component of the query. A query is resolved when all the selection attributes are resolved and a list containing the TIDs of all satisfying tuples is produced. After a query is evaluated, the query is resolved by accessing the indices specified in the query's method and performing the link to the subfiles in the order specified in the query's method. In each step of the method, the access path specified in the step is actually performed and a TID list is created of subtuples that satisfy the equality condition predicate of the attributes that are to be resolved in that step. For a conjunctive query, this TID list is intersected with the (old) TID list that is the result of the preceding steps of the resolution process. If the query is a disjunction, the union of the new TID list is taken with the old TID list. The final TID list obtained from the last step of the method is the result of the query resolution phase. In the process of query resolution, page accesses are made to secondary storage when performing an access path.

3- A query is answered when all subtuples containing projection attributes that are pointed by the TID list are retrieved into primary memory and the attribute values of attributes specified in the projection component of the query are extracted and returned. This phase of query processing involves only input/output operations and no internal processing. As previously mentioned, if the last subfile that is linked in the resolution phase

contains projection attributes, then it is possible to start answering the query before the query is completely resolved by extracting the projection attribute values from a subtuple when the selection attribute values of the subtuple satisfies the predicate. In other words, the query resolution and the query answering phases may overlap on the last subfile in the method.

In the rest of this section we will discuss the query evaluation strategy the query evaluator of our attribute partitioning system uses. Finding a satisfactory method for a query in a partitioned environment is an involved task. Unlike query evaluation in an unpartitioned environment where the query evaluator has only to choose the optimal set of applicable indices, a query evaluator in a partitioned environment in addition has to choose the sequence of linking to the subfiles. Our query evaluator is a heuristic evaluator that finds a satisfactory method for the query without resorting to cost estimation. The method obtained by the query evaluator is not necessarily the optimal method for that query, although (in the course of our work) we have found it to be near-optimal. We will first discuss the query evaluation strategy for conjunctive queries. Thereafter we discuss in what way the strategy used for disjunctive queries is different.

Query evaluation consists of two stages. In the first stage, the query evaluator selects the subset of applicable indices to include in the method. After this has been determined, the query evaluator has to choose a sequential order for linking to the subfiles that contain the rest of the selection attributes.

1- Depending on the attributes in the selection component, their selectivities, and the attribute partition, it may be beneficial to use none, all, or a subset of the applicable indices. We believe that for most queries, using either none or all of the applicable indices will lead to

satisfactory methods. Also in order to reduce the problem of query evaluation to a manageable size, we will restrict our attention to the above two choices.

One criterion by which we may judge the effectiveness of utilizing the indices to process a query is the joint selectivity of the indexed attributes that occur in the selection component of the query. Assume that: 1- the indexed attributes are not jointly selective (i.e., the joint resolving power of the indices is low and a large fraction of the tuples will be selected so that almost all the pages of a subfile that is linked thereafter have to be retrieved), and assume that 2- a subfile that contains an indexed attribute also contains some other unindexed selection attributes. Then such a subfile will most likely be accessed in its entirety in order to resolve the unindexed attribute. Therefore, the indexed attribute in the subfile can be resolved by the link at the same time the unindexed attribute is being resolved and with no extra cost. Hence when the indexed attributes are not jointly selective, using the indices will not save in the number of pages accessed.

Thus, when the joint selectivity of the indexed attributes is not too low (which is the case for the great majority of queries), the query evaluator will choose to use the full set of applicable indices. This is because the cost of resolving an attribute utilizing an index (if available) on the attribute is usually a fraction of the cost of resolving that attribute by linking to (or sequentially searching) the subfile containing it. This can be true even if the subfile containing the indexed attribute contains other unindexed selection attributes and has to be eventually linked: If the indexed attribute and the unindexed selection attributes residing in the same subfile as the indexed attribute are resolved simultaneously by linking from a TID list to their subfile, there may be more pages accessed than when the indexed attribute is resolved first using the index, the TID list pruned and reduced (as the result of

the indexing), and then the subfile linked to resolve the unindexed selection attributes.

Whether all the applicable indices are used or none of the applicable indices are used, the query evaluator will have to choose a sequence for linking to subfiles containing unresolved selection attributes. This is done in the second stage of query evaluation.

2- The second stage of query evaluation begins when the indices that are to be used have been chosen. The query evaluator will then have to link to the subfiles containing the unresolved selection attributes starting from the TID list that is the result of the indexing. Everytime a subfile containing an unresolved selection attribute is linked, the TID list is reduced to a TID list of tuples that satisfy the selection attributes in the subfile in addition to the previously resolved attributes. The subfiles containing unresolved selection attributes are linked in succession, producing successively more refined TID lists. When all the subfiles have been linked, the query is resolved and the TID list points to the selected subtuples. The task of the query evaluator in this stage of query evaluation is to find the optimal sequence of linking to subfiles. Note that the query evaluator does not actually perform the linking. The query evaluator only decides on the sequence of linking to the subfiles. It is the query resolver that actually performs the linking (in the sequence decided by the query evaluator) and retrieves the subtuples from the subfiles. The query evaluator may need to know the expected cost of linking to subfiles when deciding on the sequence. An estimate of the expected cost of linking to subfiles can be obtained without actually performing the linking. In Chapter 4 we describe the function used for this cost estimation. This function translates the number of tuple retrievals into the number of page retrievals and only requires the size of the TID list from which the linking is performed. The size of the TID list is readily

available as the product of the joint selectivity of the attributes resolved so far and the total number of subtuples in the subfile (the joint selectivity of a set of attributes is obtained by expression 3.6.1 from the individual attribute selectivities). Also note that if no index is chosen in the first stage of query evaluation, the first subfile of the sequence is sequentially searched (which is tantamount to linking to the subfile from a TID list containing all the TIDs of the subtuples in the subfile).

The criterion for optimization in this stage of query evaluation is the minimization of the total number of page accesses when answering the query. Depending on the sequence chosen in this stage, the method of a query may be optimal or highly nonoptimal. Therefore it is important that the query evaluator use a query evaluation strategy which guarantees that the sequence chosen is close to optimal for most of the queries evaluated. As we mentioned before, exhaustive enumeration of all $k!$ possible subfile sequences (where k is the number of subfiles containing unresolved selection attributes) is out of the question because cost estimating all of the sequences is computationally intractable. Due to the large search space (of possible sequences) and the numerous parameters that have to be considered in choosing a sequence, finding the optimal subfile sequence is a difficult task. However, we may qualitatively arrive at desirable sequences by considering the following criteria when deciding on the subfile sequence. 1- Subfiles that can have their selection attributes resolved without incurring too many page accesses should be linked prior to linking to subfiles that incur many page accesses. That is, at each step where a subfile is to be linked, the query resolver should link to the subfile that results in the smallest number of page accesses. Equivalently, this means linking to the subfile with the largest blocking factor (number of tuples per page), since the subfile with the largest blocking factor will result in the fewest pages accessed. (To

see this, we refer the reader to the discussion of the page access function presented in Chapter 4 and expression 4.2.1.b. In this expression, for fixed n and fixed r, A(n,b,r) monotonically decreases as b increases.) 2- The subfile that makes the joint selectivity of the resolved attributes become highest (most selective) should be linked first. That is, consider the joint selectivity of the unresolved selection attributes of each subfile and select the subfile with the highest joint selectivity (i.e. the subfile that reduces the TID list the most) to be linked next. In this manner, the overall joint selectivity will tend to become high as early as possible, causing the TID list of satisfying subtuples to be reduced earlier and fewer page accesses to be incurred as the query resolver goes to the next step of the method. The above two criteria can be conflicting requirements. A subfile may have low blocking factor but high joint selectivity for unresolved selection attributes, while another subfile may have large blocking factor but low joint selectivity.

Based upon the above criteria, we have developed five query evaluation strategies (heuristics) for choosing the subfile sequence. Each strategy is based upon one of the above criteria or uses a function of both criteria to rank the subfiles in some sequential order. Needless to say, we do not expect that any single strategy would be able to find the optimal sequence for all queries made to a database which is partitioned in any manner. However, we require that the sequence chosen by a good strategy never to be far from the optimal sequence. In order to compare the different strategies which we present, we have conducted a set of experiments on each of the strategies. In order to determine to what degree the determined strategies are optimal and to what extent they may serve the purpose of query evaluation, we have also applied the set of experiments to two other "control" strategies, and compared the results with the results of the five strategies. The five strategies considered are:

(a) Least Page Access (LPA) - In this strategy, the subfile that results in the least number of page accesses is linked. That is, when there are a number of subfiles containing unresolved attributes, the query evaluator chooses to link to the subfile that would result in the least number of page accesses. This is in accordance with the first of the two ordering criteria discussed above. Intuitively, linking the first few subfiles will result in not too many page accesses, and as the subfiles that incur many page accesses are linked further on, the joint selectivity of the attributes resolved so far will be sufficiently high such that not too many page accesses will be made to resolve the remaining attributes. As mentioned above, this strategy amounts to sequencing the subfiles according to decreasing blocking factor.

(b) Least Page Access by Pairs (LPAP) - In this strategy, the query evaluator looks at all ordered pairs of subfiles. For each pair, the query evaluator computes the cost of linking to the first subfile of the pair and adds to it the cost of subsequently linking to the second subfile. The computed cost for all the pairs is compared and the query evaluator selects the pair with the least cost to be the next two subfiles that are linked in the method. Note that when the query evaluator computes the cost of each ordered pair of subfiles, the second subfile will be linked from a subset of the TID list from which the first subfile is linked. This is because after linking to the first subfile, the TIDs of subtuples that did not satisfy the selection attributes in the first subfile are pruned from the TID list. Thereafter, the query evaluator reapplies the LPAP strategy to the remaining subfiles to select the next two subfiles that are to be linked in the method. The reapplication is repeated until all the subfiles have been sequenced. Everytime a pair of subfiles is

selected, the TID list is reduced to a smaller TID list of tuples that in addition satisfy the selection attributes of the pair of subfiles just selected.

The LPAP strategy is similar to the LPA strategy in that the criterion for sequencing is the number of page accesses. However, this strategy looks at two subfiles at a time and also considers the joint selectivity of the unresolved selection attributes of the first subfile in choosing the subfile pair. Therefore, this strategy will always result in better methods compared to the methods chosen by the LPA strategy. Observe that if there are only two subfiles that have to be sequenced in the second stage of query evaluation, then this strategy will find the optimal sequence.

(c) Highest Subfile Selectivity (HSS) - In this strategy, subfiles are sequenced according to their resolving power: The subfile containing unresolved selection attributes with highest joint selectivity is chosen to be linked first, and the subfile with the second highest joint selectivity is linked second, etc. This is in accordance with the second ordering criterion discussed above. The idea here is to reduce the size of the TID list as fast as possible.

(d) Highest Selectivity and Least Pages (HSLP) - It is desirable to order the subfiles both according to the joint selectivity of the selection attributes and according to the number of pages accessed when linking to them. The previous strategies chose one or the other as the ordering criteria. This strategy combines the two criteria by ordering the subfiles according to the (increasing) product of the joint selectivity of the selection attributes and the number of page accesses incurred in linking to the subfile; the subfile with the least product is selected and the strategy is reapplied to the remaining subfiles. Everytime the

strategy is applied to the subfiles, the subfile with the least product is selected and the number of subfiles that are to be sequenced is reduced by one. This strategy is based upon the assumption that considering both criteria will result in a superior method compared to a method that is found using a single criterion. Note that in this strategy, everytime a subfile is chosen, the TID list is reduced to reflect the resolution of the attributes in the newly chosen subfile (i.e. the joint selectivity of attributes resolved so far is multiplied by the selectivities of selection attributes in the chosen subfile). Thereafter, when choosing among the remaining subfiles, the number of page accesses incurred in linking to a subfile is computed from this reduced TID list.

(e) Highest Selectivity and Least Pages by Pairs (HSLPP) - This strategy is like the Highest Selectivity and Least Pages strategy except that all ordered pairs of subfiles are compared together. For each pair, the number of page accesses (computed in the same way as in the LPAP strategy) is multiplied by the joint resolving power of all the selection attributes in the pair of subfiles. The pair with the smallest product is chosen. The strategy is then applied to the remaining subfiles. Compared to the HSLP strategy this strategy performs a search of depth two and hence will result in superior methods than those found by the HSLP strategy.

We have conducted a number of experiments on the above five subfile sequencing strategies. The experiments varied over two different sets of query usage patterns, two partitions, three sets of attribute lengths, and three sets of attribute selectivities. The results given in the table below are the average for each strategy's performance. The two strategies Exhaust and Random are "control" strategies against which the other strategies are to be

compared. The Exhaust strategy finds the optimal sequence of subfiles by exhaustively enumerating all sequences, and selecting the sequence which results in the least processing cost for the query. The Random strategy finds a sequence for the subfiles by randomly choosing one of the possible subfile sequences, in what amounts to a non-strategy. The first row of Table 1 is the ratio of the average page accesses for each strategy with respect to the page accesses of the Exhaust strategy. The second row is the ratio with respect to the Random strategy (for the same set of experiments).

The performance of the Least Page Access by Pairs strategy was very close to the optimal performance. By the performance of a strategy we mean the cost of answering the queries in the usage pattern when each query is evaluated according to the strategy. The Least Page Access strategy also compares favorably to the other strategies. The performance of the strategies that considered the joint selectivity were not as good as the LPAP strategy. Even the LPA strategy, which only considers the number of page accesses, performed better

Exhaust	Random	LPA	LPAP	HSS	HSLP	HSLPP
1.0	1.425	1.103	1.004	1.288	1.246	1.055
0.701	1.0	0.773	0.704	0.903	0.875	0.740

Table 1 The results of different query evaluation strategies.

than the HSLP strategy that considered both the page accesses and the selectivity. We attribute this partly to the fact that after the first subfile has been linked, the joint selectivity of the resolved attributes has become high enough so that the second subfile incurs comparatively fewer page accesses than the first subfile. Thus it becomes important that the first subfile incur as few page accesses as possible.

The LPAP strategy is very close to optimal and may be considered as the choice for a query evaluator in a partitioned database environment. However in our work, we have chosen the LPA strategy because of the following reasons. 1- The LPA strategy is near-optimal. 2- The LPA strategy is computationally efficient compared to all the other strategies. Since the number of pages accessed in linking from a TID list to a subfile is inversely proportional to the subfile's blocking factor (again, refer to Section 4.2 and expression 4.2.1.b), a query evaluator based on the LPA strategy initially has to order all the subfiles of a partition according to decreasing blocking factor. For each partition, the subfiles of the partition need to be ordered only once. Thereafter when evaluating a query, the query evaluator sequences the subfiles that contain unresolved selection attributes in accordance with the precomputed sequence based upon the subfile blocking factor.

The figures of Table 1 are performance averages over different queries, partitions, attribute lengths, and attribute selectivities. Obviously, some strategies perform better than others for certain queries and partitions. It was observed that in general, as the number of attributes in the selection components of queries increases, the performance of each strategy deteriorates with respect to the Exhaust strategy, with the strategies that consider only a single subfile (the LPA, HSS, and HSLP strategies) deteriorating the most. Also, it was observed that the larger the number of subfiles in the partition, the less optimal the

performance of the various strategies.

The above discussion concerned conjunctive queries. For disjunctive queries, the query evaluation strategy used is very similar to the strategy used for conjunctive queries. If the indexed selection attributes are highly selective, and if the subfile containing the indexed selection attributes also contain unindexed selection attributes, then with great likelihood, this subfile will be searched in its entirety and using indices will not be very effective and may be avoided. Otherwise, the full set of applicable indices is used. The subfile containing unresolved selection attributes are then sequenced according to the LPA strategy (i.e. according to decreasing blocking factor). For a disjunctive query, the joint selectivity of the resolved attributes is computed according to expression 3.6.2 from the individual attribute selectivities.

A disjunctive query is resolved differently from a conjunctive query in the query resolution phase: When a TID list is obtained by linking to a subfile, the union of the new TID list is taken with the old TID list. The resulting TID list is then complemented to obtain a list of subtuple TIDs that do not satisfy any of the attributes resolved so far. This complemented TID list is used when linking to the next subfile in the method. Complementing a TID list is accomplished by repeatedly generating subfile TIDs using expression 3.2.2 and checking to see that a generated TID does not occur in the TID list.

After the query evaluation phase, the query's method is passed to the query resolver which actually produces the TID list of selected subtuples. The TID list is then used for linking to subfiles containing projection attributes. Depending on the transaction type, the query answerer does the following:

1- Query - The subfiles containing the projection attributes are linked from the TID list constructed at the query resolution phase. The selected subtuples are retrieved from the subfiles, and the values of projection attributes are extracted and returned.

2- Update - The selected subtuples are retrieved from subfiles containing projection attributes (as for a query), all attribute values to be updated are updated (in primary memory), and the subtuples are written back in their previous location. An update incurs as many page accesses as a query in the resolution phase, and twice the number of page accesses in the answering phase. If any of the updated attributes are indexed, then the affected indices are maintained as appropriate.

3- Deletion - All co-subtuples of the selected tuples are retrieved, marked deleted, and written back in their previous locations. A deletion incurs the same cost as a query in the resolution phase, and twice the cost of retrieving all co-subtuples (i.e. the entire tuple) of selected tuples in the answering phase. The affected indices are maintained as appropriate. An overflow garbage collection may ensue if there are too many deleted tuples in the file.

An insertion is different from the other transactions. Assuming that the unused tuples are uniformly scattered throughout the file, inserting r tuples in the file incurs twice the number of page accesses required for retrieving r uniformly distributed subtuples from each of the subfiles. This number is computed from the page access function of Chapter 4. If the unused tuple slots in the file have been exhausted, then the excessive inserted tuples are appended to the end of the file. In this case, the number of

page accesses incurred for each subfile will be the number of appended subtuples divided by the blocking factor of the subfile. Indices are maintained as appropriate.

We note here that the optimal attribute partition is independent of index maintenance, and the cost of maintaining the indices is incurred regardless of the choice of partition. Also we have taken the two problems of index selection and attribute partitioning as separate, assuming that the set of indexed attributes is fixed. Therefore, index maintenance cost will not enter our objective cost function, and we may eliminate it from further consideration.

6. Parameter Acquisition

The Parameter Acquisitor monitors the database management system and collects statistics both on the usage pattern and on the response of the database management system to the queries. The statistics collected are used to forecast database and usage pattern parameters for the next time interval. A time interval is the time span between two consecutive repartitioning points. The forecasted parameters will be used by the file cost estimator and the attribute partitioning heuristics at the repartitioning point marking the end of the time interval. Monitoring the database management system is a real time activity; it has to be performed while the database management system processes transactions. For this reason, only those statistics that can be inexpensively acquired should be collected. Also, the statistics collected must be succinct and require little storage for their preservation. The statistics collected for the purpose of attribute partitioning fall into four general classes:

1- Database Usage Statistics - For each query made to the database, the type of the query is stored in a table of query types. The type of a query is the set of attributes in the selection component and the set of attributes in the projection component and a flag indicating whether the query is conjunctive or disjunctive. Consequently, all queries with the same attributes (but with possibly distinct attribute values in the equality condition predicate) are clustered together in the same entry of the table. (A query type may be encoded as a bit map for the sake of succinctness.) Our assumption that the fraction of tuples satisfying an equality condition predicate depends only on the selection attributes and not on the attribute values in the selection component makes this clustering scheme possible. The number of queries that are clustered in the query type is recorded along with the query type in the table entry.

2- Average Relation Size and Average Blocking Factor - The number of tuples in each file is needed for the purpose of cost analysis. This statistic is continuously updated by the number of tuples inserted or deleted so that it reflects the instantaneous size of the file. The blocking factor of the file (the number of tuples per page) is also required for cost analysis. The blocking factor at a certain point in the time interval is the number of tuples in the file divided by the number of pages in the file at that point in the time interval. The number of pages in the file is also updated continuously as pages are allocated for inserted tuples or as pages are released after garbage collection, so that it reflects the true state of the database. Since tuples will be continuously inserted and deleted, while some tuples will be temporarily unused (until a tuple is inserted in place of a deleted tuple or until the next overflow garbage collection occurs), a fixed value for the

blocking factor over the time interval will at best reflect an average of the true blocking factor. The average blocking factor parameter is obtained by averaging the blocking factors observed at a number of points in the time interval.

3- Attribute Selectivity Statistics - This statistic is the fraction of tuples that have historically satisfied an equality condition predicate on the attribute. To compute the selectivity of an attribute, the parameter acquisitor records the number of times the attribute occurs in equality condition predicates of queries, and for each such query, the parameter acquisitor records the fraction (or an approximation thereof) of tuples that satisfied the equality condition. The average of these fractions is thus the attribute selectivity measure. Below, we describe how the fraction of selected tuples is determined.

Let σ_{ij} be the fraction of tuples that satisfy an equality condition predicate involving the ith attribute and occuring in the jth query. The attribute will be resolved by either sequential searching, indexing, or linking. If the attribute is resolved by sequential searching (i.e. the subfile containing the attribute is searched in its entirety), then σ_{ij} can be precisely calculated as the ratio of the number of tuples satisfying the equality condition predicate n, to the total number of tuples. If the attribute is resolved by indexing, then a TID list will be obtained that points to the selected tuples, and σ_{ij} is precisely the ratio of the size of the TID list to n. If the attribute is resolved by linking, then σ_{ij} has to be calculated in a reduced tuple space and then extrapolated to the entire space. This is because linking is performed from a reduced set of tuple TIDs, which have been identified beforehand, in order to get a further reduced set of tuples that additionally satisfy the predicate. Depending on whether query j is conjunctive or

disjunctive, the estimation of σ_{ij} will be done as follows.

(a) Suppose the equality condition appears in a conjunction of L equality conditions of the form:

$$C_1 \wedge C_2 \wedge \ldots\ldots \wedge C_L$$

where C_i is an equality condition involving attribute a_i. (The order of the equality conditions above reflects the order the predicates are sequenced in the query's method.) Let n_0 be the total number of tuples in the relation, and let n_i be the number of tuples that satisfy $C_1 \wedge C_2 \wedge \ldots\ldots \wedge C_i$. (Note that these numbers are readily available from the query processor when it resolves the transaction.) σ_{ij} for query j can then be approximated as:

$$\sigma_{ij} = n_i/n_{i-1}$$

(b) Suppose the equality condition appears in a disjunction of L equality conditions of the form:

$$C_1 \vee C_2 \vee \ldots\ldots \vee C_L$$

where C_i is an equality condition involving attribute a_i. (The order of the equality conditions above reflects the order the predicates are sequenced in the query's method.) Let n_0 be the total number of tuples in the relation, and let n_i be the number of tuples that satisfy $\neg C_1 \wedge \neg C_2 \wedge \ldots\ldots \wedge \neg C_{i-1} \wedge C_i$. (Again, these numbers are readily available from the query processor.) The fraction of tuples satisfying C_i of query j can then be approximated as:

$$\sigma_{ij} = n_i / (n_0 - \Sigma_{1 \le k < l} n_k)$$

The attribute selectivity s_i for attribute a_i may now be computed as the average of σ_{ij} for all $j \in Q$:

$$s_i = \frac{1}{|Q|} \Sigma_{j \in Q} \sigma_{ij}$$

where Q is the set of queries made to the database during the previous time interval. By averaging the fraction of tuples satisfying the actual occurences of an attribute in the queries, we have taken into consideration both skewness in the distribution of attribute values (for the attribute) in the file as well as skewness in the distribution of attribute value occurences in queries.

The selectivity of an attribute should change if either the distribution of its values in the file changes, or if the values in the equality condition predicates of queries involving the attribute change. Since the above changes occur when tuples are inserted, deleted, or updated and also as the database usage pattern evolves, the attribute selectivity measures need to be continuously updated to reflect the recent and more accurate information. The attribute selectivity measure is kept up to date by maintaining a running average of each attribute selectivity, as the fraction of tuples satisfying an equality condition predicate on the attribute is calculated in the process of query resolution. Every time a search is done on an attribute of a file (or subfile), the attribute's selectivity is updated by the weighted average of the old selectivity and the fraction of the tuples selected in the search.

After the individual attribute selectivities have been obtained, the joint conjunctive

or disjunctive attribute selectivities may be computed from them. Our assumption of independence among attribute value occurences in tuples leads us to simple formulas for the joint selectivities. The expected fraction of tuples that satisfy a conjunction of equality conditions simultaneously is equal to the product of the individual expected fractions that satisfy each equality condition. The joint conjunctive selectivity of a set of attributes I, each with selectivity s_i is:

(3.6.1) $$\prod_{i \in I} s_i$$

Similarly, the expected fraction of tuples that satisy a disjunction of equality conditions simultaneously is the complement of the fraction expected not to satisfy any of the equality conditions in the disjunction. The joint disjunctive selectivity for a set of attributes I, each with selectivity s_i is:

(3.6.2) $$1 - \prod_{i \in I} (1 - s_i)$$

4- The last statistical information needed is the performance cost of the partitioned database in the current time interval. This is the cost (in terms of the number of page accesses) incurred when the database management system answers all the queries in the usage pattern. This statistic is the sum total of the number of page accesses made in answering queries since the last repartitioning point. The parameter acquisitor updates this figure everytime a query is made to the database. This statistic is used to determine the extent to which the partitioned database performance cost comes close to the performance cost that had been estimated at the previous repartitioning point. If the partitioned database

performance cost is not within a reasonable distance from the performance cost that was forecasted for it, then it may be concluded that the current usage pattern no longer reflects the forecasted usage pattern and that the current attribute partition is no longer suitable. If the database performance is diagnosed as such, then a repartitioning of the database may be initiated.

When repartitioning is initiated at a repartitioning point, the parameter acquisitor takes the statistics collected in the time interval since the last repartitioning point (and also the statistics collected during previous time intervals) and forecasts parameters for the time interval up to the next repartitioning point. Specifically the parameter acquisitor forecasts the following parameters.

(a) The frequency of occurence of each query type.

(b) The size of the relation and the average blocking factor.

(c) The average selectivity of each attribute.

A thorough discussion of the exponential smoothing forecasting technique that should be used for the purpose of predicting the above set of parameters appears in [15] and [7]; we will only give an outline here. Intuitively, exponential smoothing uses a weighted moving average that is based on two sources of evidence: the most recent observation and the forecast made previously. The new forecast is equal to a percentage (known as the smoothing constant) of the recent observation plus the complement percentage of the previous forecast. Exponential smoothing has a number of advantages including simplicity of computation, minimal storage requirements, adjustability for responsiveness, and generalizability to account for trends. A variant of exponential smoothing known as adaptive

forecasting may also be used. This technique takes trends in the parameters into account. It is an efficient technique and more reliable than exponential smoothing, and may be preferred to simple exponential smoothing in some cases. For a thorough discussion of the different forecasting techniques, the reader is referred to the two works mentioned above.

7. Repartitioning Points

Database repartitioning points may be determined in several ways. Repartitioning may either be initiated by the database administrator whenever the database administrator deems necessary, or may be initiated by the parameter acquisitor. One way to have the parameter acquisitor itself initiate repartitioning is to require it to prepare at each repartitioning point a forecast of the usage pattern and the database parameters for a number of periodic checkpoints into the future. For each checkpoint, the performance cost of the partitioned database is forecasted. During the course of monitoring the database management system and the performance of the partitioned database, whenever a checkpoint is reached, the parameter acquisitor compares the observed performance cost with the performance cost forecasted at the previous repartitioning point for that checkpoint. If the observed performance is inferior to the forecasted performance by a margin that is not acceptable, the parameter acquisitor may conclude that the current partition is no longer suitable for the current usage pattern and should then initiate repartitioning. When repartitioning is initiated, forecasts of the usage pattern and database parameters are prepared for a number of periodic checkpoints into the future. (Finding the optimal set of checkpoints is itself another database optimization problem. We refer the reader to a brief

discussion of the problem of optimal determination of repartitioning points presented in Section 7.1.) The attribute partitioning heuristics are then invoked to find a suitable partition that is optimal or near-optimal for the forecasted usage pattern. If the proposed partition is different from the current partition, the current attribute partition is also cost estimated for the forecasted usage pattern. If the cost of the proposed partition is less than the cost of the current partition by a margin that justifies repartitioning, the repartitioning of the database is carried out.

CHAPTER 4

COST ANALYSIS AND THE FILE COST ESTIMATOR

In this chapter we will analyze the cost of resolving and answering a query made to the database and describe how the file cost estimator derives the total system performance cost for a given partition and a set of queries. (Henceforth, we will use the term partition performance cost to mean the performance cost of the database management system partitioned in a specified way, in response to the queries in the usage pattern. Also, we will use the term evaluating a partition to mean the derivation of the partition's performance cost by the file cost estimator.) Each of the attribute partitioning heuristics repeatedly calls upon the file cost estimator to evaluate partitions they propose. Thereafter, they select the partition with the best evaluation and based on it propose another set of partitions. Each proposed partition is cost evaluated by the file cost estimator and the partition with the best evaluation is selected. This process of deriving a set of partitions and selecting the best partition is then repeated. By this process, the heuristics try to propose partitions that result in successively better evaluations. So in a sense, the file cost estimator may be viewed as our objective cost function, which the heuristics proceed to minimize by proposing better and better partitions.

We will assume throughout that internal processing costs (CPU costs) are insignificant and the performance of the database management system we model is bounded by input/output operations (page read and writes) and hence that page accessing cost dominates all internal processing costs. Internal processing costs include the costs of query

evaluation (assumed to be negligible) and of obtaining intersections and unions of TID lists in query resolution. We assume that forming the intersection and union of TID lists can be entirely done in primary memory and so does not incur any page access to storage devices. Accessing the index of an attribute and retrieving the TID list of the index, do incur page accesses. Therefore, we include these costs in computing the partition's performance cost.

We do not consider data storage costs in the partition's evaluation. This is because if page breakage at the end of a subfile is ignored, the amount of storage required by every attribute partition of a file is the same. No matter how the file is partitioned, the storage area required for that file will be the same as that for the one-file partition plus an insignificant number of pages due to page breakage. When repartitioning an attribute partition, for each subfile at most one page can remain unfilled; the change in storage requirement from one partition to another cannot exceed the maximum number of subfiles in the two partitions. Since this figure is usually insignificant compared to the total number of pages required to store the data, we may safely ignore page breakage and hence storage costs from consideration in the evaluation of a partition.

Based upon the above assumptions, the performance cost of a partition will be the cost of accessing the subfiles in order to answer the queries in the usage pattern. Since page access cost is proportional to the number of page accesses, our cost analysis will solely be concerned with the number of page accesses incurred in answering a query. Before we discuss the file cost estimator, we give the page access analysis for each of the sequential search, linking, and indexing access paths.

1. Sequential Search

If a query's method does not specify any index to be searched (this happens if none of the query's selection attributes are indexed, or if the query evaluator deems the indices useless for resolving the query), then the first subfile of the method has to be sequentially searched in its entirety (the rest of the subfiles will be searched using links). In a sequential search, all the pages of the subfile are retrieved, and their subtuples are matched against the attribute value specified in the query selection predicates, and the TIDs of qualifying subtuples are stored in a TID list. If F_i is the subfile that is being sequentially searched, n the number of tuples in the subfile, and b_i the blocking factor for F_i, then the number of page accesses will be equal to the number of pages in the subfile:

$$\lceil n/b_i \rceil$$

The blocking factor b_i is equal to the system page size S divided by the length of the subtuple. If A_i is the set of attributes in subfile F_i, and l_j the length of attribute a_j then:

$$b_i = \lfloor S/\Sigma_{a_j \in A_i} l_j \rfloor$$

2. Tuple Retrieval Using Links

Assume that we have a list of TIDs pointing to the subtuples of a subfile. We want to compute the number of page accesses incurred in retrieving the subtuples with TIDs in the list. (Such a TID list might have been obtained either by a sequential search on a subfile, by following a previous link to another subfile, by indexing on an attribute, or by forming the intersection or union of TID lists obtained in any of the previous ways.) In any case, an

estimate of the number of TIDs in the list (which is equal to the number of tuples to which they point) is readily available from the joint selectivity of the attributes that have been resolved so far, whose resolution has resulted in this TID list. If s is the joint (conjunctive or disjunctive, depending on whether the query is conjunctive or disjunctive) selectivity of all attributes that have been considered in the creation of the TID list, and if n is the number of tuples in the subfile, then the length of the TID list is approximated by $s * n$.

Our cost criterion for performance optimization is the number of page accesses. In a paged memory environment in which tuples are blocked together in pages, we have to translate the expected number of tuple accesses to the expected number of page accesses. The expected number of pages to be accessed is always less than or equal to the number of tuples to be accessed because two or more tuples may reside on the same page. In our model of the database management system, finding the expected number of page accesses is relatively easy because of the following properties, which hold as a result of the assumptions we have made about our file and index models:

1- The TID list is ordered. Whether the TID list is obtained by sequential searching, linking, or indexing it is ordered (i.e. sorted in increasing or decreasing value) and subsequent intersections and unions preserve this ordering. This property of the TID list assures that each page of a subfile is retrieved at most once (since the tuples are retrieved in the sequence they reside in the subfile. This property also eliminates the need for large buffer areas in primary memory to accomodate input/output operations, since at any instance, at most one page will be in primary memory.)

2- The TID list is not redundant; i.e., no TID appears more than once in the list.

3- The TIDs are distributed uniformly over the TID space. This property is assured by our assumption in Section 3.4 of independence among the occurence of attribute values in tuples. Hence all TIDs appear with equal probability in the TID list, and the tuples they point to are scattered uniformly throughout the subfile.

These three properties of our model of tuple access makes the translation of the number of tuple accesses to page accesses relatively simple. Based upon these three properties, Yue and Wong [40] have derived the number of page accesses from the number of tuple accesses in terms of the recurrence relation 4.2.1.

(4.2.1.a) $$A(n,b,0) = 0$$

(4.2.1.b) $$A(n,b,r+1) = \frac{n-r-b}{n-r} A(n,b,r) + \frac{n}{n-r}$$

In the above formula, A(n,b,r) is the expected number of pages accessed from a file (subfile) with n tuples (subtuples) and b tuples (subtuples) per page when retrieving r tuples (subtuples). (Note that $r = s * n$ in the cost analysis above.) The computation of 4.2.1 involves on the order of r multiplications and r divisions, and is therefore quite expensive to compute. By the technique of generating functions, we have solved the recurrence relation 4.2.1 and have obtained the closed form solution 4.2.2 for the number of page accesses.

$$A(n,b,r) = \frac{n}{b}\left[1 - \frac{\binom{n-r}{b}}{\binom{n}{b}}\right] \qquad (4.2.2)$$

The above formulation has the advantage over the recurrence relation that it can be computed more efficiently. A detailed derivation of the formulation 4.2.2 (hereafter called the page access function) may be found in Appendix I of [7]. (Waters [36] and Yao [37] have also independently arrived at the page access function using the hypergeometric distribution of probability theory.) The formulation 4.2.2 also admits of a simple interpretation. For an arbitrary page in the file, the probability that it does not contain any of the r selected tuples is the number of ways of choosing b tuples from $n - r$ tuples, divided by the number of ways of choosing b tuples from n tuples. Hence the expected number of page accesses will be the number of pages (n/b) times the complement of the above probability.

During the course of attribute partitioning, the attribute partitioning heuristics repeatedly call upon the file cost estimator to evaluate partitions. Every time the file cost estimator evaluates a partition, it has to estimate the cost of answering each of the query types in the table of query types. Estimating the cost of answering each query type involves computing the number of page accesses incurred in accessing each of the subfiles that contains an attribute in the selection or projection components of the query type. Since for each such subfile, we have to compute the page access function, it is important that the page access function be computed as efficiently as possible. The page access function 4.2.2, if expanded, will take on the order of $\min(b,r)$ multiplications per computation. Although the page access function is much more efficient in computation than the recurrence relation 4.2.1

(since b is usually much smaller than r), computing the page access function in its exact form 4.2.2 is still too costly for our purposes. Instead, we use the following approximation to the page access function (suggested by Michael Hammer) in our file cost estimations:

$$A(n,b,r) \cong \frac{n}{b}\left[1 - e^{\,b \log\left(1 - \frac{r}{n-(b-1)/2}\right)}\right] \qquad (4.2.3)$$

The approximation 4.2.3 has proven to be very fast, taking only a constant number of multiplications and divisions per computation, and has the advantage of extreme accuracy for almost every combination of n, b, and r [8].

3. Index Accessing and Tuple Retrieval

Using the index of an attribute of a file (or subfile), in order to retrieve tuples that have a given value for that attribute, is composed of three steps. The first step is accessing the non-leaf pages of the index to get a pointer to the TID list of tuples with the given attribute value. The second step consists of retrieving this TID list. The third step is retrieving the tuples that the TIDs point to by retrieving the pages they reside in the subfile. A detailed analysis of indexing costs appears in Chan [7] and we will not reproduce it. We shall only repeat here the final expression derived in [7]. The average cost of using an index is:

$$\lceil \log_{u_n S/(1+L)} [(n + L/s)/u_l S] \rceil + \lceil (s * n + L)/u_l S \rceil + A(n, b, s * n)$$

where n = number of tuples in the file

b = blocking factor of the indexed attribute's subfile

L = length of the indexed attribute

s = selectivity of the indexed attribute

u_n = average fraction of index node page utilization

u_l = average fraction of index leaf page utilization

S = system page size.

The three terms of the expression are the respective costs of the three indexing steps. The last step of index use, i.e. retrieving the qualifying tuples, actually occurs if this attribute is the only one whose index is used in the method of the query. In all other cases, the intersection or the union of the TID list obtained from the second step of indexing is taken with other TID lists before the tuples that are pointed to are retrieved.

4. File Cost Estimation

The file cost estimator evaluates a partition proposed by the partitioning heuristics and computes the performance cost for that partition. The performance cost of each proposed partition is estimated by iterating over the queries in the table of query types and estimating the cost of answering each query. (This table is provided by the parameter acquisitor and is a forecast of the database usage pattern for the next time interval.) Each query type in the table is passed for evaluation to the query evaluator. The query evaluator uses the Least Page Access strategy and thereby produces a near optimal method for the

query. (The Least Page Access strategy, as described in Section 3.5, sequences the subfiles that are to be linked according to decreasing subfile blocking factor.)

The file cost estimator receives the method for the query. If any index is specified to be accessed by the method, the file cost estimator uses the cost expression for index use to compute the cost of accessing the indices. If no index is specified by the query's method, the first subfile in the method has to be sequentially searched and the cost of the search is the number of pages in the subfile. (The reason that the first subfile must be sequentially searched is that initially there is no TID list on hand that would restrict the search to certain pages of the subfile. Sequential searching may be viewed as a limiting form of linking, where each page of the subfile has to be retrieved.) In either case, i.e. if indices are used or the subfile is sequentially searched, the joint selectivity of the attributes resolved so far can be readily computed from expressions 3.6.1/3.6.2, depending on whether the query is conjunctive or disjunctive. (The set of attributes I in 3.6.1 and 3.6.2 is the set of attributes resolved so far.) The remaining subfiles of the method are sequenced and are to be linked in the sequence specified by the method. Using the approximation to the page access function, the file cost estimator computes the cost of accessing the first of these subfiles. (Observe that r, the number of tuples to be retrieved, equals the product of the joint selectivity of attributes resolved so far and the number of tuples in the subfile n). The access cost estimated for this subfile is then added to the cost of indexing/sequential searching. The file cost estimator then derives the new joint selectivity figure by including the old joint selectivity figure and the selectivities of all the attributes resolved in this step of the method in expression 3.6.1/3.6.2. The cost of linking to the remaining subfiles of the method in the sequence specified is then computed for each successive subfile in the same way. The cost of accessing each subfile is

added to the accumulated cost, and the joint selectivity is updated according to the selectivities of the newly resolved attributes. When no subfile remains in the method, the file cost estimator will have computed the cost of resolving the query and also the joint selectivity of the query (and hence the number of tuples selected by the query). The file cost estimator then computes the cost of answering the query. Using the approximation to the page access function and the estimate of the number of tuples that are selected by the query, the file cost estimator estimates the number of pages that need to be retrieved from each subfile that contains any of the projection attributes. The only subfile containing a projection attribute that does not incur page retrievals in the answering phase is the last subfile in the method of the query. This is because the projection attributes in this subfile can be retrieved as the selection attributes of this subfile are being resolved. The cost accumulated in the resolving and answering phases is then summed to give the cost estimate for the query. A query's cost estimate is then multiplied by the frequency of the query type to get the total cost estimate for that query type. Finally, the sum of these weighted query cost estimates is the performance cost of the partition (in the context of the forecasted usage pattern).

The file cost estimator is called repeatedly in the process of attribute partitioning. It is imperative that the file cost estimator be implemented efficiently. Note that by clustering all queries with the same type into one entry of the query type table, we have already reduced the totality of the queries in the usage pattern into a relatively smaller set of query types. Hence, the number of the iterations required by the file cost estimator has already been reduced. Although further clustering measures like the "nearest centroid" clustering scheme of Belford [5] could be employed to still reduce the number of query types, the degree of query clustering we have employed has proven sufficient for our purposes. Tests show that

the file cost estimator (as programmed in the programming language MDL [26] on a PDP-10) takes somewhat less than a second of processing time to estimate the cost of a set of 100 conjunctive and disjunctive query types. Furthermore, when queries are additionally clustered, correlation information about attribute occurences in queries will be inevitably lost, and estimates based on clustered queries will be less reliable. Therefore further query clustering is not advisable.

5. Repartitioning Cost

The cost of repartitioning the attribute partition is computed as follows: if at a repartitioning point, the new partition has subfiles $F_{k+1}, \ldots, F_M$ in common with the old partition, then only subfiles $F_1, \ldots, F_k$ have to be retrieved, reorganized, and written back on secondary storage. The total page accesses required to do this will be twice the number of pages in each subfile:

$$2 \sum_{i=1,\ldots,k} \lceil n/b_i \rceil$$

The above cost is based on repartitioning all the subfiles $F_1, \ldots, F_k$ simultaneously. I.e., the pages of each subfile are read in sequence along with the pages of the other subfiles, the attribute values are then transferred from one subfile to another, and finally the pages are written back onto secondary storage. Each page of a subfile is thus accessed only twice, once for reading and once for writing.

At each repartitioning point, the performance cost of the partition proposed by the partitioning heuristics for the next time interval has already been computed. The performance cost for the current file partition for the next time period is then computed. The

two are compared and if the proposed partition offers a performance cost reduction greater than the cost of repartitioning, then the file should be reorganized according to the proposed partition.

CHAPTER 5

THE ATTRIBUTE PARTITIONING HEURISTICS

In this chapter we present a number of heuristics for partitioning the attributes of a file. Each attribute partitioning heuristic starts with a supplied partition and derives from it a superior partition. (If the heuristic is not able to improve on the supplied partition, the heuristic will terminate and return the supplied partition as its result.) Therefore it is possible to apply the attribute partitioning heuristics in succession, with each heuristic starting with the resultant partition of the preceding one and producing a partition that is as good as the preceding partition. We say that a heuristic is relevant to a partition if its application will result in an improved partition.

We have performed a number of experiments on the attribute partitioning heuristics. The overall results of the experimentation performed on each heuristic is included in the discussion of that heuristic. Since our most extensive program of experimentation was applied to our main heuristics, we have devoted Section 7 of the chapter entirely to a detailed discussion of that subject. Before we proceed to describe the heuristics, we will first establish the necessary terminology for the subsequent sections.

Let P be a partition of the set of attributes A of a file into disjoint subsets. Each subset of A is termed a block of attributes; the ith block of the partition is denoted by A_i. A block of attributes may be viewed as a representation of a subfile; i.e., when a file is partitioned according to a given partition P, each block A_i of P is directly implemented by

a subfile with attributes drawn form A_i. If M is the number of blocks in the partition, then $P = \{A_i\}_{i=1}^{M}$, $A_i \cap A_j = \phi$ for all $i \neq j$, and $\cup_{i=1}^{M} A_i = A$. The trivial partition P^0 has been defined previously to be the partition where every subfile contains exactly one attribute. That is, $P^0 = \{A_i^0\}_{i=1}^{m}$, where $A_i^0 = \{a_i\}$.

1. The Exhaustive Enumeration Approach

One way of finding the optimal partition is to produce all partitions of the set of attributes, and evaluate each of them with the file cost estimator in order to identify the partition with the best performance cost. This exhaustive enumeration approach is not a viable partitioning strategy because of the large number of possible ways to partition a file. The number of distinct partitions of a set of m elements into disjoint subsets, $B(m)$, is known as the mth Bell number. Unfortunately, there is no simple expression for $B(m)$ that we can analyze in order to arrive at its complexity. However, Moser and Wyman [27] provide an asymptotic expansion for the Bell numbers. This asymptotic expansion is in terms of the solution to the equation $x e^x = m$, and hence is not in closed form. From this asymptotic expansion it is possible to derive the following asymptotic upper and lower bounds for the Bell numbers [28]:

(5.1.1) $$B(m) = o(m^m)$$

(5.1.2) $$m^{(1-\epsilon)m} = o(B(m)), \qquad \epsilon > 0$$

where ϵ is any non-zero positive real number. (The notation $f(m) = o(g(m))$ denotes that $\lim_{m\to\infty} f(m)/g(m) = 0$.) The two asymptotic bounds are very tight, and from them we see that the number of partitions of a set of m elements into disjoint subsets asymptotically

approaches close to m^m (or equivalently, close to $2^{m \log_2 m}$) as m approaches infinity. By this we mean that ϵ may be taken as small as possible as long as it is positive, and $B(m)$ will always grow faster than $m^{(1-\epsilon)m}$. Therefore for all practical purposes, the number of distinct attribute partitions is prohibitively high to render an exhaustive enumeration approach feasible (for the general attribute partitioning problem with any number of attributes). As an example, a file containing 10 attributes can be partitioned into $B(10) = 115975$ different partitions. Another problem with the exhaustive enumeration approach is that generating all the $B(m)$ different partitions is not an easy task. A program written for generating all the partitions of a set of attributes (and which was used to exhaustively find the optimal partition for a number of attribute partitioning problems with not more than 8 attributes) required storage space that grew faster than $B(m)$.

2. The Stepwise Minimization Heuristic

The heuristics we have considered in our work and described in subsequent sections are all stepwise minimization heuristics. Stepwise minimization is the process of carrying out an optimization task in a series of steps. At each step, a cost criterion is optimized to the extent possible. Each step that follows carries the optimization still further. Finally, when no further optimization can be performed at a step, the stepwise minimization process is terminated. In the case of the attribute partitioning heuristics, each heuristic starts from a predetermined partition, and in each step tries to come up with a new partition that is an improvement over the partition of the preceding step. By improvement we mean that the performance cost of the improved partition, as evaluated by the file cost estimator, is less than

the performance cost of the previous partition. Once an improved partition is found at a step, the next step starts with the newly found partition and tries to find a still better partition. This process of incremental improvement is continued until no partition may be found which is an improvement to the partition considered in the last step of the heuristic. The last partition is then returned by the heuristic as the resultant partition of the heuristic. The intermediate partition found at each step of the attribute partitioning heuristics will depend on the partition of the last step, the query frequencies, and the query types.

At each step of the attribute partitioning heuristics we have considered, the improved partition is obtained from the partition of the previous step by either 1- grouping a number of blocks of the last partition together to form a single block, or by 2- degrouping a block of the previous partition into two or more blocks. The heuristics we have considered differ from one another in two respects: 1- the attribute partition that they initially start with, and 2- the manner in which the blocks are grouped or degrouped in each step.

In our work, we apply a heuristic to an initial partition until in the course of stepwise minimization, the heuristic produces a partition upon which it can no longer improve. At this point we may apply a second heuristic to the resultant partition of the first heuristic. After the application of the second heuristic, a third heuristic may be applied, or even the first heuristic may be reapplied. Since a heuristic always results in a partition that is as good as the partition that it starts with, it is always possible to apply any number of heuristics in succession and never get a partition with a higher performance cost (and occasionally get an improved partition). However, some of the heuristics we consider are best succeeded by certain other heuristics. In the discussion of each heuristic, we will make it clear if the heuristic performed well enough to warrant further investigation, and if so, what other

heuristics were tried in combination with it.

Note that one mode of operation we do not consider is trying a heuristic for only one or a few steps and switching to another heuristic before the first heuristic produces its final resultant partition. Our mode of applying the attribute partitioning heuristics is based upon the assumption that if a second heuristic is relevant to an intermediate partition produced by a first heuristic (that is, the second heuristic can improve upon the performance cost of the intermediate partition), then the second heuristic will still be relevant after the first heuristic has terminated. This assumption is made in order to reduce to a manageable size the problem of deciding which heuristic to apply next.

We shall consider a number of heuristics in the forthcoming sections. However, the pairwise grouping heuristic described in the next section is the main heuristic of this work and we will attempt to describe it in full detail. In our experimentation, we have found that the combination of the pairwise grouping heuristic with a second heuristic (the single attribute degrouping-regrouping heuristic) to be sufficient for the purpose of attribute partitioning within the context of the database management system we have considered.

3. The Pairwise Grouping Heuristic

The pairwise grouping heuristic begins with the trivial partition P^0, and generates all partitions that can be obtained by grouping together pairs of blocks in P^0. For example, if $A = \{1, 2, 3, 4\}$ are the attributes of a file, the pairwise grouping heuristic begins with the trivial partition of row 0 of Figure 1 and produces all the partitions of row 1 of the same figure. The heuristic then evaluates all the generated partitions with the file cost estimator,

and finds the partition (call it P^1) whose performance cost is the least of all the generated partitions. In other words, assume $C(P)$ to be the performance cost of partition P as determined by the file cost estimator. In the first step of the heuristic, the following minimization is performed:

(5.3.1) $$\min_{1 \le j < k \le m} C(P^1_{j,k})$$

where $P^1_{j,k} = \{A^0_1, ..., A^0_j \cup A^0_k, ..., A^0_m\}$. Let j and k be the values that minimize 5.3.1. If it is the case that $C(P^1_{j,k}) < C(P^0)$, then the improved partition $P^1_{j,k}$ is the result of the first step, and the second step of the heuristic begins with partition $P^1 = P^1_{j,k}$. Otherwise, if it is the case that $C(P^1_{j,k}) \ge C(P^0)$, the heuristic terminates (with the trivial partition as the resultant partition). In general, the ith step of the pairwise grouping heuristic starts with partition $P^{i-1} = \{A^{i-1}_1, A^{i-1}_2, ..., A^{i-1}_{M_{i-1}}\}$ (where M_{i-1} is the number of blocks in P^{i-1}), and performs the minimization:

(5.3.2) $$\min_{1 \le j < k \le M_{i-1}} C(P^i_{j,k})$$

where $P^i_{j,k} = \{A^{i-1}_1, ..., A^{i-1}_j \cup A^{i-1}_k, ..., A^{i-1}_{M_{i-1}}\}$. Assuming j and k minimize 5.3.2, and if $C(P^i_{j,k}) < C(P^{i-1})$, the heuristic then goes to step $i+1$ starting with $P^i = P^i_{j,k}$, $M_i = M_{i-1} - 1$. This process is continued until a step (say step L) is reached for which $C(P^L_{j,k}) \ge C(P^{L-1})$ for all j and k. At this point, no pair of blocks can be found that grouping them will reduce the performance cost, and so P^{L-1} is returned as the result of the pairwise grouping heuristic.

The pairwise grouping heuristic may be depicted in terms of a lattice where each node of the lattice is a partition. The top node is the trivial partition and the bottom node is the "one-file" partition. An interior node is obtained by grouping together a pair of blocks of one of its parents. Figure 1 shows such a lattice for the set of four attributes {1, 2, 3, 4}

(from here on we shall use integers to represent attributes). The ith row of the lattice corresponds to all the partitions that could be generated by the ith step of the pairwise grouping heuristic (equivalently, all the possible partitions with which the i+1th step may begin). The pairwise grouping heuristic begins with the trivial partition and produces all the partitions that can be reached by following an edge (i.e., all the partitions of the first row). It then selects the partition in that row with the best performance cost. From that partition, it follows all the edges leading downwards to its children nodes. For example if the second partition from the left is the best partition of row 1, then in the next step the heuristic

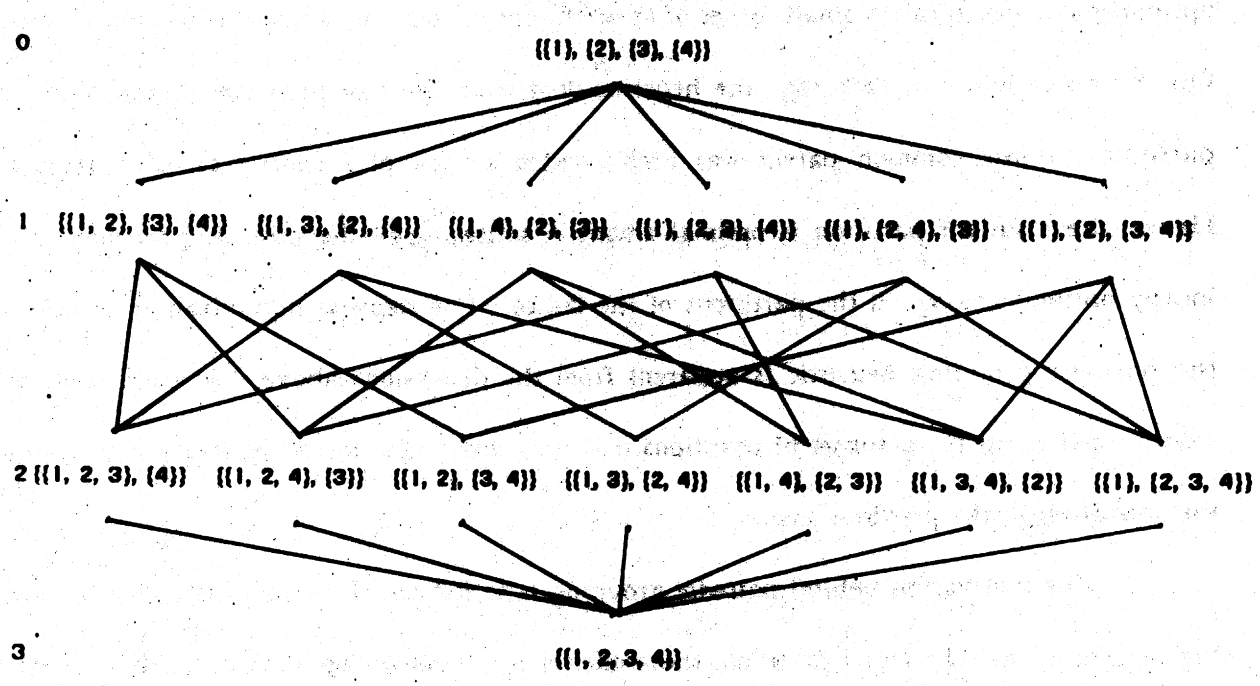

Figure 1 The lattice of partitions.

would compare the first, fourth, and sixth partitions of row 2. The three partitions of the second row are all, in a sense, desirable partitions in that they have been derived from a partition that has previously been proved superior. Specifically, the heuristic assumes that the optimal partition is either among the three partitions or is somewhere below them in the lattice and can be reached by going down the edge from the best of the three partitions. The heuristic continues to go down the lattice until none of the partitions examined in a row reduce the performance cost. At this point, the current parent partition is returned as the resultant partition of the pairwise grouping heuristic.

The resultant partition of the pairwise grouping heuristic is not necessarily the optimal partition. Only a small subset of all partitions are actually examined by this process. On the other hand, at each step, the heuristic does select the best partition among a set of partitions, whose common parent was itself selected as best of a similar set of partitions. Hence, the resultant partition is optimal among a subset of all the possible partitions and locally optimal among all the partitions of the lattice. The stepwise minimization nature of the pairwise grouping heuristic is apparent from the discussions above. At each step, we minimize the cost for a subset of partitions that have been selected on the basis of a similar minimization in the previous step.

The motivation behind pairwise grouping is as follows: Initially, when all attributes are separated in the trivial partition, those queries that request two attributes are answered with close to minimum cost, while those queries requesting more than two attributes are very costly to answer because their attributes reside in different subfiles and hence on different pages. Subsequently, as blocks of attributes are grouped, queries requesting a small number of attributes become costlier to answer because accessing the attributes will bring in those

attributes that are not requested by the query but nevertheless reside in the same subfile as a requested attribute, while those queries requesting many attributes, of which all or some are in the same subfile, become less costly to answer. In the process of grouping blocks together, a point will be reached where the reduction in cost of answering those queries that are benefited by the grouping will not offset the increase in cost of answering those queries that become costlier due to the grouping. This point is a local minimum of the performance cost function.

The Bond Energy Algorithm of McCormick et al. [24] is another stepwise minimization heuristic that may be used for the purpose of attribute partitioning. Hoffer and Severance [19] have used the Bond Energy Algorithm to group attributes into blocks based on the similarities of attribute occurences in queries (see Chapter 2 for a detailed discussion of how Hoffer and Severance [19] utilize the Bond Energy Algorithm for the purpose of attribute partitioning). We believe that our pairwise grouping heuristic, when compared to the Bond Energy Algorithm, has a number of advantages which makes it more desirable as a partitioning vehicle. The Bond Energy Algorithm operates by permuting the columns of a matrix consisting of pairwise attribute access similarity measures in such a way that the columns of similar attributes fall close together. If we look at the matrix of pairwise access similarity measures after the algorithm has terminated, we will find that the attributes are ordered such that similar attributes are placed adjacent or nearly adjacent to one another. A disadvantage of this algorithm is that after this is accomplished, i.e. after such an ordering of the attributes is found, it is left to subjective judgement to decide how to clump the attributes together to form blocks. The other disadvantage of this algorithm (and one which will be examined in Section 4) is that the algorithm only looks at the similarity of access between

pairs of attributes (i.e. between pairs of blocks, each of one attribute) rather than among any number of blocks containing any number of attributes.

Stepwise minimization is basically a hill climbing heuristic search technique which will not necessarily locate the optimal solution. The solution achieved using this technique may be any of the local minima; the closeness of the solution to the optimal partition will depend on the database parameters, the access paths of the file, and the usage pattern parameters. However, in the course of our experimentation with the pairwise grouping heuristic (to be described in full detail in Section 5.7), the pairwise grouping heuristic starting with the trivial partition has consistently resulted in either the optimal partition or in a near-optimal partition that differed insignificantly from the optimal partition. This has led us to believe that pairwise grouping is an attractive heuristic search technique for finding an adequate partition for the attribute partitioning problem.

The process of pairwise grouping is actually the method of steepest descent of the hill climbing heuristic search technique. The coordinates of a point on the "hill" (which should be visualized as inverted, since the search is for finding the minimum point) are the partition and the performance cost of the partition as determined by the file cost estimator. The distance between two partitions is defined as the number of edges on the minimum path connecting the two partitions in the lattice of partitions. Pairwise grouping is the process of following the negative gradient from one point to an adjacent point with a distance of one (along the partition axis), beginning at the point of the trivial partition. Our conclusion from this program of experimentation has been that this "hill" is predominantly devoid of "bumps" (i.e. local minima or points where the gradient changes sign and all adjacent points to the "bump" have a larger performance cost). The few "bumps" that occur on the hill

happen to be very close to the top of the "hill" in such a way that these "bumps" are not significantly lower than the top of the "hill" itself (the global minima). In other words, it appears that the "hill" which is the performance cost of the partitioned database is rather "smooth" and the "bumps" that indicate partitions that are locally minimum all occur near the highest point of the "hill".

As we will fully report in Section 5.7, the performance of the pairwise grouping heuristic is extremely good, managing to improve the performance of the database management system by a factor of 5 with respect to the one-file partition (i.e., the unpartitioned file). The resultant partition of the pairwise grouping heuristic was in all cases either optimal or near optimal. To determine the extent to which the partition produced by the pairwise grouping heuristic is optimal (by comparing it with the optimal partition), we exhaustively produced the optimal partition for cases where m was less than or equal to 8 ($B(8) = 4140$). For cases where m was greater than 8, we either manually generated promising partitions, or applied other heuristics to the resultant partition of the pairwise grouping heuristic. In all cases, the resultant partition of the pairwise grouping heuristic was either the optimal partition, or was within a few percent of it. In the few cases where the resultant partition did not coincide with the optimal partition, it was observed that some blocks in the resultant partition contained one or more spurious attributes that, if removed from their blocks, improved the performance. These spurious attributes, it turned out, were inserted in their corresponding blocks rather early in the minimization process. That is, although a spurious attribute resulted in performance improvement early in the grouping process (in other words a spurious attribute was attractive to its block), later on, as more attributes were added to the same block, the spurious attribute was no longer attractive to the

augmented block, and degrouping it from its block or transferring it to other blocks became beneficial. In Section 6 we will describe other stepwise minimization heuristics, some intended to eliminate this deficiency of the pairwise grouping heuristic.

The major disadvantage of the pairwise grouping heuristic is its slowness. Every partition that the heuristic produces by grouping a pair of blocks needs to be cost evaluated by the file cost estimator. Let us see, in the course of carrying out the minimization 5.3.2, how many partitions need to be cost estimated. In the analysis below, we have assumed that if the file has m attributes, then on the average the heuristic will iterate for $\lceil m/2 \rceil$ steps. This is a reasonable assumption in that the heuristic may iterate anywhere from 1 to $m - 1$ steps. (Our analysis will still hold as long as the number of steps is a constant fraction of m.) The improved partition produced at a step has exactly one less block than the partition of the previous step, i.e. $M_i = M_{i-1} - 1$, and the ith step will start with a partition that has $M_{i-1} = m - i + 1$ blocks. Under this assumption, the final partition produced by the heuristic has $\lfloor m/2 \rfloor + 1$ blocks: there are $\lceil m/2 \rceil - 1$ steps that produce an improvement and a final step that produces no improvement. Therefore at the ith step, there will be binomial(m-i+1, 2) partitions generated that need to be cost estimated. On the average, the total number of file cost estimations will be:

$$\sum_{i=1}^{\lceil m/2 \rceil} \binom{m-i+1}{2}$$

The above expression is of order m^3. This order is quite large, and it is desirable to reduce the partition search cost by doing either of the following two things: 1- reduce the cost of file cost estimation by evaluating partitions in some other manner, or 2- reduce the number of file cost estimations by cost estimating only some of the partitions generated at each step of

the heuristic. The next section will investigate the possibility of cost reduction by employing means other than file cost estimation. Section 5 will investigate the possibility of reducing the number of partitions that need to be cost estimated.

4. File Cost Estimation as a Measure of Block Attractivity

Assume we have the partition $P = \{A_1, ..., A_M\}$. Define the (n-wise) block attractivity measure α of $\{A_{j_1}, ..., A_{j_n}\} \subseteq P$ to be the cost reduction (or increase) obtained by grouping the n blocks together:

$$\alpha(\{A_{j_1}, ..., A_{j_n}\}; P) = C(P) - C(\{A_1, ..., A_{j_1} \cup ... \cup A_{j_n}, ..., A_M\})$$

If $\alpha(\{A_{j_1}, ..., A_{j_n}\}; P)$ is positive, then partition P may be improved by grouping $A_{j_1}, ..., A_{j_k}$. If $\alpha(\{A_{j_1}, ..., A_{j_n}\}; P)$ is negative, then grouping $\{A_{j_1}, ..., A_{j_n}\}$ will increase the performance cost. The pairwise grouping heuristic cost estimated all partitions P_{jk} generated by grouping together blocks A_j and A_k of P in the ith step, and selected the partition with minimum cost. Since C(P) was available from the previous step, cost estimating all P_{jk} where $1 \le j < k \le M_{i-1}$ is equivalent to finding all the block attractivities $\alpha(\{A_j, A_k\}; P)$, and minimizing $C(P_{jk})$ is equivalent to finding the maximum value for $\alpha(\{A_j, A_k\}; P)$ where $1 \le j < k \le M_{i-1}$. The two blocks A_j and A_k that maximize α are then grouped in going from the current step to the next step. (They are grouped only if $\alpha(\{A_j, A_k\}; P) > 0$.) From this discussion, we see that a step of pairwise grouping may be viewed as the process of finding the pairs of blocks in a partition with maximum attractivity.

In the previous section we noted that one way the pairwise grouping heuristic may be made more efficient is by refraining from file cost estimation. That is, by computing the

block attractivity measure for all pairs of blocks without resort to file cost estimation.

In order to avoid computing $\propto$, we may try to find an approximation to it that is easily computable. An approximate block attractivity measure must possess certain qualities of the (ideal) block attractivity measure $\propto$ in order to guarantee that the outcome of the pairwise grouping heuristic using the approximate measure is the same as the outcome using the (ideal) block attractivity measure (the reason we call $\propto$ the ideal measure is that no other measure can be better than true file cost estimation). Below we give a number of properties that a candidate measure for the approximation to the block attractivity measure must necessarily possess

1- The candidate measure should depend not only on the set of blocks in the partition that their attractivity is being computed, but also on how the rest of the attributes are blocked in the partition. That is, the same set of blocks may have a different attractivity depending on the partition they are in. Because of this, we just cannot compute the attractivity measure of a pair of blocks by considering only the pair of blocks and their attributes. Rather, attractivities must be computed in the context of the partition as a whole. Consequently, the attractivity measure of a pair of blocks in a step of the pairwise grouping heuristic depends also on the partition reached at that step. (If this were not the case and the candidate measure depended only on the pair of blocks, then it will become possible to redistribute the attributes in the rest of the blocks such that the ideal measure changes and becomes of a different sign signifying that the pair of blocks have become attractive or unattractive, while the candidate measure remains unchanged.) It is for this reason that the attractivities computed in a step of the pairwise grouping heuristic cannot be used in future steps.

2- The candidate measure must be computed by considering all queries made to the file.

3- The candidate measure should be applicable to blocks with any number of attributes in them, rather than just being applicable to blocks containing a single attribute.

4- It must be possible to compute the attractivity of any number of blocks; i.e., the candidate measure must be an n-wise attractivity measure. We need this capability because some of the other heuristics we shall consider require the mutual attractivity of more than two blocks. One might assume that the attractivity of a set of more than two blocks can be obtained by applying the pairwise measure to all pairs of block in the set, and making the n-wise measure the sum total of the pairwise measures. This is not true because the pairwise attractivity measure is not transitive. For example, if blocks A_1 and A_2 are attractive, and likewise blocks A_2 and A_3, we cannot conclude that it is desirable to block A_1, A_2, and A_3 into the same block, because blocks A_1 and A_3 may be unattractive to each other. Even if it were the case that all pairs of A_1, A_2, and A_3 were attractive, there is no reason to conclude that all three blocks are mutually attractive. Hence, an approximate measure must also be an n-wise block attractivity measure. (Note that the ideal block attractivity measure is an n-wise measure.)

We have considered the attribute similarity measure of Hoffer and Severance [19] as a candidate for the approximate block attractivity measure. The pairwise attribute similarity measure of two attributes (or equivalently two blocks, each containing one attribute) is the ratio of the information transferred (from secondary storage to primary memory, for the purpose of selecting tuples that satisfy a predicate or for the purpose of projecting a value), to the total information transferred from a subfile consisting solely of the two attributes (when

answering queries that request one or both of the attributes). In other words, assume that the two attributes exist by themselves in one subfile. When a query that request one or both of the attributes is made to the database, a portion (for some queries all) of the subfile's subtuples will be transferred to primary memory (depending on the joint selectivity of the other attributes of the query that have already been resolved), and only a certain percentage of this portion will actually be useful data (because the first of the two attributes may further restrict the set of tuples to the tuples that satisfy a predicate on the attribute). This percentage is the ratio discussed above. There is one such ratio for each query type. The weighted average of all such ratios will be the pairwise attribute similarity measure of the two attributes (the weight of a query type is its frequency). From this definition we see that the pairwise attribute similarity measure applies to only pairs of blocks, where each block consists of only one attribute. (Hoffer [4] describes an extension of the attribute similarity measure that works for more than two blocks, but where each block still consists of one attribute only.) The other problem with the attribute access similarity measure is that it does not depend on the other blocks in the partition; it just depends on the two attributes of the measure. For the above reasons, we have concluded that the attribute access similarity measure of Hoffer and Severance cannot be used as an approximate measure of the block attractivities.

Considering the above prerequisites for an approximate measure, and the fact that the file cost estimator implemented turned out to be rather fast in estimating the performance cost of a partition (the speed of the file cost estimator is partly due to the rather simple query evaluation strategy that it uses), we decided to retain the ideal measure (i.e. the difference between the performance costs of the old and new partitions) for the purpose of finding the most attractive pair of blocks in the pairwise grouping heuristic. Using the ideal block

attractivity measure, we hope to bring out the real characteristics of the attribute partitioning problem, characteristics that might be blurred and obscured using a less than ideal measure. Indeed, the choice of the file cost estimator as the attractivity measure has proven useful to us and has allowed us to considerably reduce the number of pairwise attractivity measures that need to be computed. The next section elaborates on how this reduction is achieved.

5. The Fast Pairwise Grouping Heuristic

In the previous section it was pointed out that the attractivity of a set of blocks in a partition is also dependant on the other blocks of the partition. However, in the course of our experimentation, it was observed that if blocks A_1 and A_2 were attractive to one another at a step of the pairwise grouping heuristic, then A_1 and A_2 remained attractive with almost the same attractivity measure at the next step, with one possible exception. The exception is when either of blocks A_1 or A_2 were grouped (together or with another block) in going from one step to the next step. In terms of the attractivity measure, we observed that:

(5.5.1) $$\alpha(\{A_x, A_y\}; P) \cong \alpha(\{A_x, A_y\}; P_{jk})$$

or equivalently that:

$$C(P) - C(P_{xy}) \cong C(P_{jk}) - C(P_{jk,xy}) \qquad \text{for all } x \neq j, k \text{ and } y \neq j, k$$

where $P = \{A_i\}_{i=1}^{M}$, $P_{jk} = \{A_1, \ldots, A_j \cup A_k, \ldots, A_{M-1}\}$, $P_{xy} = \{A_1, \ldots, A_x \cup A_y, \ldots, A_{M-1}\}$, and $P_{jk,xy} = \{A_1, \ldots, A_j \cup A_k, \ldots, A_x \cup A_y, \ldots, A_{M-2}\}$. It was also observed that the attractivity measures of pairs of blocks retained their relative order (in terms of magnitude) with respect to one

another, from one step of the pairwise grouping heuristic to the next step, even when the pairwise attractivity measure values were not exactly the same as the last step. In short, no pair of blocks A_1 and A_2 were found that were attractive in one step and were not attractive with almost the same attractivity in subsequent steps of which A_1 and A_2 remained ungrouped.

There is no intrinsic property of the database management system we have modelled that implies observation [illegible] of 5.5.1 (i.e., that a pair of blocks should have radically different attractivity in going from one step to the next step of the pairwise grouping heuristic) can come about only inferentially because of the following reasons:

1. Let $C(P; q)$ be the cost of answering query q when q is made to the database partitioned according to P (by definition we have $\sum_{q \in Q} C(P; q) = C(P)$, where Q is the usage pattern). In order that 5.5.1 be violated, $C(P; q) - C(P_{xy}; q)$ should be different when A_1 and A_2 are grouped from when A_1 and A_2 are not grouped. Let us call such a query q where $C(P; q) - C(P_{xy}; q) \neq C(P_{jk}; q) - C(P_{jk,xy}; q)$ a volatile query. In order for 5.5.1 to be violated, not only that the volatile query have to be highly significant (that is, $C(P; q) - C(P_{xy}; q)$ should be significant compared to $C(A_1, A_2, P)$, where q is the volatile query), but also if there happens to be more than one volatile query in the usage pattern, the effect of these volatile queries should not be to cancel each other out.

2. It is not sufficient that a query contain an attribute that is in A_1, A_2, A_3 or A_4 in order to be volatile. One necessary condition for a query to be volatile is that the query requests an attribute from $A_1 \cup A_2$ and also an attribute from $A_3 \cup A_4$. If the query does not

request an attribute from $A_j \cup A_k$, then $C(P_{jk}) = C(P)$ and $C(P_{jk,xy}) = C(P_{xy})$ and the query is non-volatile. If the query does not request an attribute from $A_x \cup A_y$, then $C(P_{xy}) = C(P)$ and $C(P_{jk,xy}) = C(P_{jk})$ and the query is non-volatile. Another necessary condition for a query to be volatile is that grouping A_j and A_k should make a difference in the processing of the volatile query. That is, a volatile query must be processed with a different method in partition P (or P_{xy}) compared to the method it is processed in partition P_{jk} (or $P_{jk,xy}$). If this were not the case and the query's method was the same for both P and P_{jk}, then the effect that grouping A_x and A_y would have in the cost of processing the query will be the same whether or not A_j and A_k are grouped, and hence will cancel one another out in 5.5.1. This will cause the query to become non-volatile. Therefore for a query to be volatile, it must not only request attributes from both $A_j \cup A_k$ and $A_x \cup A_y$ in its selection component, but grouping A_j and A_k should influence the method in which the query is processed. From this discussion we may conclude that the queries in the database usage pattern are predominantly non-volatile.

To reiterate, the observation 5.5.1 is because the queries in the usage pattern are predominantly non-volatile, and if a query happens to be volatile, it does not significantly determine the attractivity of A_x and A_y.

Observation 5.5.1 has a number of important implications. First, it says that the pairwise attractivity measure needs to be computed for all pairs of blocks only once, and that the attractivity computed remains valid in subsequent steps as long as neither blocks of the pair participates in a grouping. Second, it says that the attractivities which need to be

the sorted list that contain either one or both of the blocks participating in the grouping. It then computes the pairwise attractivity measures between the newly created block and the rest of the blocks in the partition. The newly computed attractivity measures are then sorted and merged with the sorted list. The most attractive pair of blocks is then selected from the top of the list and grouped for the current step. The above process is then repeated on the new partition.

Our experimentation with the fast pairwise grouping heuristic has been very successful. The fast pairwise grouping heuristic consequently produced the same final partition as the pairwise grouping heuristic produced in all the tests performed. Furthermore, it was drastically more efficient than the pairwise grouping heuristic. The extent to which the fast pairwise grouping heuristic is more efficient then the pairwise grouping heuristic increases as m increases and as the number of steps the heuristic iterates increases (the two heuristics iterate for the same number of steps since they produce the same final partition). In our experiments, the fast pairwise grouping heuristic was anywhere between 1.6 to 5 times more efficient than the pairwise grouping heuristic. Because of its advantages, we have adopted the fast pairwise grouping heuristic instead of the pairwise grouping heuristic, as one of the two main heuristics of our attribute partitioning system.

6. Other Variants of the Stepwise Minimization Heuristic

In addition to the pairwise grouping heuristic described above, a number of other heuristics were developed and subsequently programmed and tested. We will describe the following heuristics:

1- the k-partition pairwise grouping heuristic,

2- the two most attractive pairwise grouping heuristic,

3- the triplewise grouping heuristic,

4- the binary degrouping heuristic,

5- the single attribute degrouping heuristic,

6- the double attribute degrouping heuristic,

7- the single attribute degrouping-regrouping heuristic,

8- the double attribute degrouping-regrouping heuristic,

9- the single attribute ungrouping heuristic.

We will also give the overall results of experiments conducted in different environments, so that the cost effectiveness of each heuristic (i.e. the optimality of their resultant partition when measured against the effort required to produce it) may be assessed. Heuristic 1 - 4 and 9 are intended as alternatives to the (fast) pairwise grouping heuristic, while heuristics 5 - 8 (which degroup blocks in each step) are intended to improve upon the resultant partition of the (fast) pairwise grouping heuristic.

1- The k-partition pairwise grouping heuristic - This heuristic is an extension of the pairwise grouping heuristic. In the first step of this heuristic, the k partitions with the best performance costs are selected and passed to the next step. Each subsequent step receives k partitions and generates all possible partitions that can be obtained from them by pairwise grouping of blocks. Out of the set of all partitions generated, the best k are selected and the process above is repeated. The pairwise grouping heuristic is a special case of the k-partition pairwise grouping heuristic where $k = 1$. Therefore, the k-partition pairwise grouping heuristic will always result in a partition that is at least as optimal as the resultant partition

of the pairwise grouping heuristic. The drawback of this heuristic is its search cost, which is roughly k times that of the pairwise grouping heuristic. In view of the near optimality (and in most cases the optimality) of the pairwise grouping heuristic, it is not possible to justify the additional search cost that this heuristic incurs.

2- The two most attractive pairwise grouping heuristic - We noted in Section 4 that block attractivities remain relatively unchanged from one step of the pairwise grouping heuristic to another if none of the blocks participate in a grouping. We also noted that the attractivity of pairs of blocks retained their relative order from one step to another. Hence, in each step of the pairwise grouping heuristic, it is likely that the few most attractive pairs of blocks will be eventually grouped in subsequent steps, and therefore we can deduce that the two most attractive pairs of blocks in a step are good candidates for being grouped simultaneously in one step.

Having this in mind, the two most attractive pairwise grouping heuristic was developed. This heuristic takes the top two most attractive pairs of blocks and groups both. pairs in one step. (If the top two most attractive pairs of blocks have a block in common, then all three blocks will be grouped in one step.) Each subsequent steps takes the partition of the last step and simultaneously groups the two most attractive pairs of blocks. Grouping the two most attractive pairs of blocks has the advantage that the number of steps required to get the final partition is reduced by a factor of 2. However, this heuristic has the disadvantage that as the number of attractive pairs diminishes towards the end, grouping the top two pairs becomes too arbitrary a thing to do: If the first best pair of blocks and the second best pair of blocks have any block in common, then it is possible that the uncommon block in the second pair is not attractive to the new block formed by grouping the first pair

(as noted previously the pairwise attractivity measure is not transitive). Another disadvantage with this heuristic is that even if the two top pairs of blocks have no common blocks, it is possible that after grouping the first best pair, another block in the partition manifests a greater attractivity to the newly created block than the attractivity of the second best pair of blocks. Indeed, tests show that the two most attractive pairwise grouping heuristic produces the same partitions in the first few steps as does the pairwise grouping heuristic, but later on it diverged and produced a less optimal partition.

3- The triplewise grouping heuristic - This heuristic is the third variation of the pairwise grouping heuristic we have considered. This heuristic groups triples of blocks at each step instead of pairs of blocks. In the pairwise grouping heuristic, we generated all the partitions by grouping pairs of blocks. The triplewise grouping heuristic generates all partitions by grouping triples of blocks and then selects the partition with best performance. A number of partitioning problems were solved using this heuristic. In all cases, the triplewise grouping heuristic produced a nonoptimal partition, and in almost every case, the resultant partition was less optimal than the pairwise grouping heuristic partition.

The inferior performance of this heuristic may be attributed to the large number of blocks it tries to group in one step. Clumping three blocks simultaneously in one step is somewhat too coarse an action. For example, in some experiments we observed a set of three blocks A_1, A_2, and A_3 that were all pairwise attractive with respect to one another, but if A_1 and A_2 were grouped together, A_3 was no longer attractive to the block $A_1 \cup A_2$ (i.e. $\alpha(\{A_1, A_2\}; P) > 0$ and $\alpha(\{A_1, A_3\}; P) > 0$ and $\alpha(\{A_2, A_3\}; P) > 0$, but $\alpha(\{A_1, A_2, A_3\}; P) \leq 0$). The triplewise grouping heuristic also suffers from the drawbacks of the two most attractive pairwise grouping heuristic mentioned above. Finally the triplewise grouping heuristic is on

the order of m slower than the pairwise grouping heuristic, in that it requires binomial(m-2 i+2, 3) attractivities to be computed in step i.

4- The binary degrouping heuristic - This heuristic resembles a logarithmic search. It starts with the one-file partition and tries to divide it into two blocks so that the performance cost improves. After this is accomplished, the heuristic tries to divide each of the blocks into a pair of smaller blocks and so on. Dividing a block is accomplished by finding all attributes that if excluded from the block, result in a reduction of the performance cost (i.e. finding all the attributes in the block that are unattractive to it). At each step, a maximum of half of these unattractive attributes are degrouped from the block into a separate block of their own. Although this heuristic is very fast, in that it is logarithmic, it performed quite poorly, and tended to produce a trivial partition at the end. One may speculate that this heuristic can be useful for a usage pattern where the optimal partition consists of 2 or 3 blocks.

5- The single attribute degrouping heuristic - This heuristic along with all the remaining heuristics are degrouping heuristics. A degrouping heuristic takes a partition, and separates out one or more attributes from one block of the partition. In terms of the lattice model of stepwise minimization, the degrouping heuristics amount to going up an edge, from a partition in one row of the lattice to another partition in the row directly above. Except for heuristic 9, all the degrouping heuristics are intended to be used to improve upon the partition that has already been arrived at by the process of pairwise grouping. By using the fast pairwise grouping heuristic and the degrouping heuristics alternately, we try to further optimize the resultant partition. When the pairwise grouping heuristic and the degrouping heuristics are applied alternately, the process of grouping and degrouping blocks may be seen

improvement. Again, experiments produced negative results and in none of the cases tried did it improve on the partition produced by the pairwise grouping heuristic.

7- The single attribute degrouping-regrouping heuristic - This heuristic eliminates the drawbacks of the previous degrouping heuristics by degrouping an attribute from its block and simultaneously grouping it with another block, all in one step. This strategy is based on the reasoning that an attribute which has been grouped fairly early in the pairwise grouping process must have had a high attractivity in the first place. It is unlikely that this attribute should exist by itself in a separate block. More likely, in the later steps of pairwise grouping, it becomes more attractive to other blocks than to its current block. Indeed, in the few cases where the pairwise grouping partition was not optimal (as compared to the optimal partition found by exhaustive enumeration), the optimal partition could have been obtained from the pairwise grouping partition by transfering one or more attributes from one block to another. In terms of the lattice of attribute partitions, this amounts to a traversal from a partition of one row to another partition of the same row by going up an edge and coming back down another edge.

The single attribute degrouping-regrouping heuristic first selects the attributes that reside in blocks of two or more attributes. For such an attributes, it computes the attractivity measure of the attribute with all the blocks in the partition except for the block that the attribute currently resides in. It then selects the attribute that is most attractive to some other block and groups it in that block. If this improves the performance cost over the performance cost of the current partition, the heuristic iterates on the improved partition. In the cases where the pairwise grouping partition did not coincide with the optimal partition, the single attribute degrouping-regrouping heuristic when applied to the resultant partition

of the pairwise grouping heuristic, resulted in the optimal partition. (Observe that if applied to the optimal partition, the single attribute degrouping-regrouping heuristic will immediately terminate and return the optimal partition.) It is difficult to determine the single attribute degrouping-regrouping heuristic's search cost. All we know is that in all cases where the heuristic was applied, it did not require more than three steps to conclude.

The fast pairwise grouping heuristic in combination with the single attribute degrouping-regrouping heuristic has proven superior to all the other heuristics as a result of our investigations. Consequently, we have selected the fast-pairwise-grouping heuristic in conjunction with the single attribute degrouping-regrouping heuristic as the choice of the partitioning heuristics for our attribute partitioning system.

8- The double attribute degrouping-regrouping heuristic : This heuristic is similar to the single attribute degrouping-regrouping heuristic except that it attempts to degroup a pair of attributes from the same block and group them both back with some other block. This heuristic has the advantage over the single attribute degrouping-regrouping heuristic that degrouping-regrouping any one of two attributes in one step may not result in an improved partition, while degrouping-regrouping both the attributes in one step may produce an improvement. In none of our experiments did we run into such a situation where double attribute degrouping-regrouping produced an improvement while single attribute degrouping-regrouping did not produce any improvement. Also, this heuristic suffers from being too coarse, as was the case with the triplewise grouping heuristic. If degrouping-regrouping just a single attribute is beneficial, this heuristic will not detect that. Hence we have dropped this heuristic from further consideration in favor of the single attribute degrouping-regrouping heuristic.

9- The last heuristic that we developed and tested is the single attribute ungrouping heuristic. This heuristic is a degrouping heuristic in that it attempts to degroup attributes from their blocks. The single attribute ungrouping heuristic performs the functions of both the single attribute degrouping-regrouping heuristic and the single attribute degrouping heuristic in one step, and it is thus superior to both of them: When an attribute is degrouped from a block in a step, it may be regrouped back with other blocks, or placed in its own block. In terms of the lattice of attribute partitions, (some of the) partitions reachable from the last partition by either an upward edge (to directly one row above, i.e. degrouping), or all partitions that belong to the same row and are reachable by following an edge up and another edge down (degrouping-regrouping) are considered.

We mentioned previously that the single attribute degrouping heuristic does not turn out to be advantageous if applied to the resultant partition of the pairwise grouping heuristic. Hence, if applied as a sequel to the pairwise grouping heuristic, the single attribute ungrouping heuristic will only be as good as the single attribute degrouping-regrouping heuristic. The single attribute ungrouping heuristic is thus meant as a stand alone heuristic: It is initially applied to the one-file partition; the minimization process moves upward from the bottom of the lattice until the optimal partition or a near optimal partition is reached. In this sense, the single attribute ungrouping heuristic is the inverse process of the combination of the pairwise grouping heuristic with the single attribute degrouping-regrouping heuristic.

Experimentation with this heuristic has generally been satisfactory. In the smaller attribute partitioning problems (in terms of the number of attributes), the single attribute ungrouping heuristic has been able to locate the optimal partition. In the larger problems, the heuristic did not perform as well as the pairwise grouping heuristic, and resulted in a less

optimal partition.

The drawback of the single attribute ungrouping heuristic is mainly its slowness in reaching a locally optimal partition. This is due to the following two reasons. 1- At each step of the heuristic, all the attributes in non-singleton blocks not only have to be degrouped, but at each step these attributes have to be regrouped with all of the remaining blocks. It is hard to assess the order of search cost for this heuristic, but it appears that it takes as many partition evaluations as the pairwise grouping and the single attribute degrouping-regrouping heuristics combined. 2- In most partitioning problems considered, the optimal partition was much closer, both in terms of the number of edges in the lattice of partitions and in terms of performance cost, to the trivial partition than to the one-file partition. I.e. the distance, in terms of the number of edges, of the optimal partition was much smaller to the top of the lattice than to the bottom of the lattice. This points to the additional advantage that a stepwise minimization heuristic that starts from the trivial partition has over a heuristic that starts from the one-file partition.

7. Experimenting with the Pairwise Grouping Heuristics

The effectiveness of our approach to attribute partitioning in a self-organizing database system can be fully evaluated only by employing it over an extended period of time in an operational setting. However, it is possible to obtain some partial assessments of these techniques by means of experimentation in a controlled environment.

We have conducted an extensive program of experimentation with the pairwise grouping heuristic, the fast pairwise grouping heuristic, and the single attribute

degrouping-regrouping heuristic. We have also performed a number of tests on the other attribute partitioning heuristics. The overall result for each heuristic was reported in the discussion of that heuristic. In general, none of the attribute partitioning heuristics performed as well as the (fast) pairwise grouping heuristic, and none of the degrouping heuristics performed as well as the single attribute degrouping-regrouping heuristic. Henceforth we will concentrate on providing a detailed description of the experimentation performed on our main heuristics and provide an assessment of the performance of these heuristics in comparison with the exhaustive enumeration procedure. The exhaustive enumeration procedure performs an exhaustive search of the full space of possible partitions and so is guaranteed to locate the true optimal partition.

Each experiment was concerned with selecting an optimal partition for a particular database in the context of some given usage pattern. In each experiment, the pairwise grouping heuristic and the fast pairwise grouping heuristic were tried starting with the trivial partition. Thereafter the single attribute degrouping-regrouping heuristic was tried starting with the resultant partition of the pairwise grouping heuristics (i.e. the pairwise grouping heuristic and the fast pairwise grouping heuristic). If the single attribute degrouping-regrouping heuristic was relevant at that point, then the pairwise grouping heuristics were retried starting with the resultant partition of the single attribute degrouping-regrouping heuristic. In addition, using the exhaustive enumeration procedure, the optimal partition was found for the same set of database parameters and the database usage pattern.

In an operational setting, the usage pattern submitted to an attribute partitioning system would be based on historical records of database use; in our experiments, we generated

these patterns ourselves. Naturally, a full program of authentication would require performing experiments for the full spectrum of data characteristics and usage patterns; since this was obviously infeasible, we had to restrict our studies to a limited set of cases. Selecting these cases posed a real dilemma. We did not want to choose them at random, for we might thereby focus on uninteresting or unrepresentative examples; but if we selected them too carefully, we would potentially sacrifice some of the validity and generality of our testing procedure. In consequence, we settled on the following approach. For each experiment, we provided a set of database parameters and a set of other parameters that roughly and succinctly characterized a particular kind of usage pattern. These succinct usage pattern parameters were then used to drive a usage pattern generator, probabilistic in nature; it randomly generated a set of individual transactions in accordance with the given parameters, and then summarized them in a usage pattern description. The output of this generator was then used as the usage pattern for the experiment.

In all our experiments, we have kept two of the database parameters fixed: n, the number of tuples in the file, was set to 100,000 and S, the page size, was set to 1024 words. We believe these figures reflect typical values for real databases; in any event, the behaviour of our attribute partitioning system should not differ radically for problems with different values for n and S. Also, we are permitted to constrain at least one database parameter without reducing the degrees of freedom inherent in our problem.

The results of our experimentation are depicted in Tables 1 through 9. Our most extensive series of experiments were conducted with files of 8 and 22 attributes. The limiting factor in our experimentation was the use of the exhaustive enumeration procedure. Even for a file with 8 attributes, the number of possible partitions in the search space is 4140 and

using the exhaustive enumeration procedure to find the optimal partition for the problem is a nontrivial task, one that can only be accomplished with difficulty. For problems with $m > 8$, we have not been able to use the exhaustive enumeration procedure to verify the performance of our attribute partitioning heuristics. However, we can reach a number of conclusions concerning the behaviour of the heuristics for problems with large m by extending the results obtained from problems with $m \leq 8$.

Table 1 shows the characteristics of our experimentation. Each row shows the number of experiments carried out for an attribute partitioning problem with 5, 6, 7, 8, 15, 22, or 30 attributes. The average number of conjunctive query types and the average number of

Number of attributes	Number of experiments	Average number of conjunctions	Average number of disjunctions	Average one-file partition cost [1]	Average trivial partition cost [1]
5	10	8	5	1.0	0.708
6	10	8	4	1.0	0.449
7	14	21	6	1.0	0.184
8	20	21	10	1.0	0.499
15	10	45	19	1.0	0.189
22	20	70	12	1.0	0.122
30	6	60	25	1.0	0.505

Table 1 Characteristics of the experiments.

disjunctive query types for each row is depicted in columns 3 and 4 of the table. The last column of Table 1 shows the performance cost of the trivial partition with respect to the one-file (unpartitioned) performance cost. That is, for attribute partitioning problems with a certain m, the ratio of the cost of the trivial partition to the cost of the unpartitioned file was computed, and column 6 of Table 1 shows the average of such ratios for each m. In the remaining tables, all columns superscripted by 'r' indicate that the figures in the column is the average of the ratios with respect to the one-file partition.

Table 2 shows the result of experiments with m = 5. The optimal partition found by exhaustive enumeration and the partition found by the pairwise grouping heuristic and the fast pairwise grouping heuristic coincided in all of the ten experiments (as indicated in column 3). The average cost of the optimal partition found with respect to the one-file partition was 0.59. The heuristics iterated on the average for 2.5 steps. This includes the last step where the heuristics could not improve upon the last partition and returned the partition found by the previous step.

Number of attributes: 5
Number of experiments: 10

	Average cost of best partition found	Number of steps heuristic iterated	Number of times optimal partition was not found
Exhaustive enumeration	0.592	-	-
Pairwise grouping heuristic	0.592	2.5	0
Fast pairwise grouping heuristic	0.592	2.5	0

Table 2 Result of experiments with m = 5.

Tables 3 and 4 depict the results of experiments with $m = 6$ and 7. Again in all the cases tested, the optimal partition was found by both the pairwise grouping heuristic and the fast pairwise grouping heuristic.

Table 5a shows the result of the 20 experiments performed with $m = 8$. As indicated in column 3, the pairwise grouping heuristic and the fast pairwise grouping heuristic found

Number of attributes: 6
Number of experiments: 10

	Average cost of best partition found	Number of steps heuristic iterated	Number of times optimal partition was not found
Exhaustive enumeration	0.434	-	-
Pairwise grouping heuristic	0.434	2.1	0
Fast pairwise grouping heuristic	0.434	2.1	0

Table 3 Result of experiments with $m = 6$.

Number of attributes: 7
Number of experiments: 14

	Average cost of best partition found	Number of steps heuristic iterated	Number of times optimal partition was not found
Exhaustive enumeration	0.175	-	-
Pairwise grouping heuristic	0.175	3.0	0
Fast pairwise grouping heuristic	0.175	3.0	0

Table 4 Result of experiments with $m = 7$.

Number of attributes: 8
Number of experiments: 20

	Average cost of best partition found [1]	Number of steps heuristic iterated	Number of times optimal partition was not found
Exhaustive enumeration	0.4657	-	-
Pairwise grouping heuristic	0.4657	4.6	1
Fast pairwise grouping heuristic	0.4657	4.6	1

Table 5a Result of experiments with $n = 8$.

Number of attributes: 8
Number of experiments: 1

	Average cost of best partition found [2]	Number of steps heuristic iterated	Number of times optimal partition was not found
Single attribute degrouping-regrouping heuristic	0.9999	2.0	0

Table 5b Result of the single attribute degrouping-regrouping heuristic with $n = 8$.

the optimal partition in all but one case. Table 5b shows the result of applying the single attribute degrouping-regrouping heuristic to the resultant partition of the pairwise grouping heuristics in the one problem where the pairwise grouping heuristics did not find the optimal partition. The single attribute degrouping-regrouping heuristic found the optimal partition in 2 steps (where the second step did not find an improvement to the result of the first step). The ratio of the resultant partition of the single attribute degrouping-regrouping heuristic to that of the pairwise grouping heuristics was 0.9999. I.e., the local minimum found by the the

pairwise grouping heuristics was within 0.0001 of the optimal solution. (Columns superscripted with '2' is the performance cost with respect to the performance cost of the partition found by the pairwise grouping heuristics in the previous table.)

Table 6 shows the results for problems with $m = 15$. In this series of experiments and in all the experiments with larger m, the exhaustive enumeration procedure had become infeasible (for example, $B(15) \cong 1.38 * 10^9$). However in all the experiments with $m = 15$, the pairwise grouping heuristic and the fast pairwise grouping heuristic coincided in their resultant partition. The average cost of the best partition found by the heuristic was 0.15 of the cost of the one-file partition for this set of experiments. Also, none of the degrouping heuristics were relevant at this stage.

Tables 7a and 7b show the result of 20 experiments with $m = 22$. In half of the experiments, it was possible to improve upon the result of the pairwise grouping heuristic (which always coincided with the result of the fast pairwise grouping heuristic). The single attribute degrouping-regrouping heuristic was applied in these 10 cases, and the partition

Number of attributes: 15
Number of experiments: 10

	Average cost of best partition found [1]	Number of steps heuristic iterated	Number of times a better partition was found
Exhaustive enumeration	-	-	-
Pairwise grouping heuristic	0.1501	8.3	0
Fast pairwise grouping heuristic	0.1501	8.3	0

Table 6 Result of experiments with $m = 15$.

Number of attributes: 22
Number of experiments: 20

	Average cost of best partition found [1]	Number of steps heuristic iterated	Number of times a better partition was found
Exhaustive enumeration	-	-	-
Pairwise grouping heuristic	0.1010	10.6	10
Fast pairwise grouping heuristic	0.1010	10.6	10

Table 7a Result of experiments with m = 22.

Number of attributes: 22
Number of experiments: 10

	Average cost of best partition found [2]	Number of steps heuristic iterated	Number of times a better partition was found
Single attribute degrouping-regrouping heuristic	0.9973	2.8	0

Table 7b Result of the single attribute degrouping-regrouping heuristic with m = 22.

reached by the single attribute degrouping-regrouping heuristic was always such that no other (pairwise grouping or degrouping) heuristic could improve upon it. The single attribute degrouping-regrouping heuristic iterated for an average of 2.8 steps. The performance cost of the partition found using the single attribute degrouping-regrouping heuristic was on the average 0.0027 less than that found by the pairwise grouping heuristics.

Tables 8a, 8b, and 8c show the result of 8 experiments with m = 30. In most of the experiments, the single attribute degrouping-regrouping heuristic was able to improve an

Number of attributes: 30
Number of experiments: 6

	Average cost of best partition found[1]	Number of steps heuristic iterated	Number of times a better partition was found
Exhaustive enumeration	-	-	-
Pairwise grouping heuristic	0.30570	21.2	5
Fast pairwise grouping heuristic	0.30575	21.0	5

Table 8a Result of experiments with m = 30.

Number of attributes: 30
Number of experiments: 5

	Average cost of best partition found[2]	Number of steps heuristic iterated	Number of times a better partition was found
Single attribute degrouping-regrouping heuristic	0.9929	2.3	2

Table 8b Result of the single attribute degrouping-regrouping heuristic with m = 30.

Number of attributes: 30
Number of experiments: 2

	Average cost of best partition found[3]	Number of steps heuristic iterated	Number of times a better partition was found
Pairwise grouping heuristic	0.9995	2.0	0
Fast pairwise grouping heuristic	0.9995	2.0	0

Table 8c Result of reapplying the pairwise grouping heuristic with m = 30.

average of 0.0071 on the partition produced by the pairwise grouping heuristics. In 2 out of 5 cases where the single attribute degrouping-regrouping heuristic was applied, the pairwise grouping heuristics when reapplied starting with the final partition of the single attribute degrouping-regrouping heuristic, produced a partition with an average further improvement of 0.0005 in performance cost. No other heuristic could improve on the partition found by the reapplication of the pairwise grouping heuristics. The column superscripted by '3' in Table 8c indicates that the figures in the column is the percentage improvement with respect to the partition found by the single attribute degrouping-regrouping heuristic in Table 8b.

Table 9 depicts the execution time statistics. For all of the heuristics considered, the number of partitions evaluated increased as the number of attributes of the file increased. The exhaustive enumeration procedure had the fastest rate of increase while the pairwise grouping heuristic had a drastically-lower rate. The fast pairwise grouping heuristic had an even lower rate of increase than the pairwise grouping heuristic in the number of partitions required for evaluation. (As we showed in the previous sections, the number of partition evaluations for the exhaustive enumeration procedure is approximately on the order of m^m, for the pairwise grouping heuristic on the order of m^3, and for the fast pairwise grouping heuristic on the order of m^2.) With $m = 30$, the fast pairwise grouping heuristic required roughly 1/5 the number of partition evaluations required by the exhaustive enumeration procedure with $m = 8$. This is a significant improvement. Note that the processing time required by a heuristic depends both on the number of partitions evaluated and also on the number of query types in the usage pattern. For example, the processing time for the fast pairwise grouping heuristic is on the order of the product of the number of query types in the usage pattern and the square of the number of attributes in the file.

Exhaustive enumeration:

Number of attributes	B(m)	Average processing time in seconds
5	52	6.8
6	203	28.5
7	877	258
8	4140	2212
15	$\approx 1.38 * 10^9$	-
22	$\approx 4.50 * 10^{15}$	-
30	$\approx 8.46 * 10^{22}$	-

Pairwise grouping heuristic:

Number of attributes	Average number of partitions evaluated	Average processing time in seconds
5	19	3.6
6	29	4.9
7	48	14.8
8	80	26.6
15	502	341
22	1521	1334
30	3964	5244

Fast pairwise grouping heuristic:

Number of attributes	Average number of partitions evaluated	Average processing time in seconds
5	16	2.2
6	21	3.6
7	31	8.9
8	44	16.3
15	175	135
22	397	382
30	791	1099

Table 9 Execution time statistics.

The result of these experiments may be summarized as follows: For problems where $m \leq 8$, both the pairwise grouping heuristic and the fast pairwise grouping heuristic found the optimal partition in all but one case. The optimal partition found was significantly superior to the one-file partition in all of the cases tested. In the one case where the pairwise grouping heuristics did not find the optimal partition, the partition that they found was within 0.0001 of the optimal partition. In this case, the single attribute degrouping-regrouping heuristic when starting with the final partition of the pairwise grouping heuristics found the optimal partition by the transfer of only a single attribute from one subfile to another. Altogether, for $m \leq 8$ where the optimal partition could be determined by exhaustive enumeration, the combination of a pairwise grouping heuristic and the single attribute degrouping-regrouping heuristic was always able to find the optimal partition. For problems where $m > 8$, it was not possible to directly verify the outcome of the pairwise grouping heuristics by exhaustive enumeration. However in most of the problems tried, the pairwise grouping heuristic found a partition that no other degrouping heuristic could improve upon. In addition, the partition found by the pairwise grouping heuristics was always significantly superior to the one-file partition. In less than half the experiments with $m > 8$, the single attribute degrouping-regrouping heuristic improved on the resultant partition of the pairwise grouping heuristic; and in these cases, the improved partition differed insignificantly from the resultant partition of the pairwise grouping heuristics, attesting to the desirability and the near-optimality of the pairwise grouping heuristics. In the very few cases where the pairwise grouping heuristics were relevant to the partition found by the single attribute degrouping-regrouping heuristic, the corresponding improvement in the performance cost was negligible. All this indicates that the combination

of a pairwise grouping heuristic and the single attribute degrouping-regrouping heuristic (when applied alternately) converge rapidly to a desirable solution. If we can extend the results for cases where $m \leq 8$ to the cases where $m > 8$, we may conclude that the optimal partition does not differ at all or does not differ significantly in performance cost from the resultant partition of the combination of a pairwise grouping heuristic and the single attribute degrouping-regrouping heuristic.

A number of observations were also made during the experimentation concerning the behaviour of the attribute partitioning system. Specifically, we comment on the following:

1- A number of experiments were performed with database parameters and usage patterns that were variations of the parameters of an earlier experiment. That is, we took the database parameters and the usage pattern of a problem and we changed the database parameters, such as the set of available indices and the attribute lengths and selectivities, or we changed the frequency and/or structure of the query types. (In all these variations the number of attributes, the number of tuples, and the page size were kept constant.) We then applied the attribute partitioning heuristics (and the exhaustive enumeration procedure, when possible) to these variations to observe how the optimal partition differed form one problem to another. As it turns out, changing the database parameters such as the attribute lengths and selectivities, sometimes even radically, did not significantly alter the optimal partition: at most one or two attributes moved from one subfile to another. However, changing the usage pattern, such as the frequency of query types or the composition of the components of the query types, sometimes drastically altered the optimal partition. Hence it appears that in our environment, the optimal

partition is more sensitive to the usage pattern and less sensitive to the database parameters, and thus that the usage pattern is more a determinant of the optimal partition than the other parameters.

2- It was observed that the larger the number of available indices for an attribute partitioning problem, the larger the subfiles of the optimal partition (i.e., more attributes were clustered together in the same subfile), and the smaller the number of subfiles in the optimal partition. In addition, the performance cost of the optimal partition (and also that of the trivial partition) became closer to the performance cost of the one-file partition as the number of available indices were increased. We provide the following as an explanation for this observation: If an attribute is indexed, the index will most likely be used to resolve that attribute when the attribute is requested by the selection component of a query. If the attribute is not indexed, then the attribute has to be resolved by either sequentially searching or by linking, and any other attribute that exists with this attribute in its subfile will also have to be retrieved in its entirety. This will incur extra page accesses. Even if the other co-existing attribute was requested in the selection component of the query, it is always easier to link to the co-existing attribute (from a reduced TID list) in order to resolve it rather than sequentially search the co-existing attribute in its entirety or link to it from a larger TID list. Therefore, having an index available on an attribute will in most cases eliminate the need for its sequential searching or linking to the subfile containing the attribute, and hence, increase the overall attractivity of the attribute to the other attributes in the file. Therefore, indices contribute to the clustering of attributes in the file. This point will again be taken up in Section 6.2 when we discuss

the relative efficacy of the trivial partition.

3- In Table 8a we see that the average cost of the partition found by the pairwise grouping heuristic is not exactly the same as that for the fast pairwise grouping heuristic. It turns out that for one of the experiments with $m = 30$, the outcome of the two pairwise grouping heuristics did not coincide. Although it is not impossible that the results of the two heuristics be different, in Section 5.5 we argued that this is highly improbable. However, it appears that this becomes more probable as m increases because the heuristics have to iterate for more steps and the pairwise attractivity measures computed by the fast pairwise grouping heuristic in the first step becomes less likely to hold as the heuristic iterates for a large number of steps. This was the case in the one problem mentioned above. I.e., a pair of attributes, neither which participated in a grouping, turned out to be unattractive in the latter steps of the pairwise grouping heuristic. Unfortunately it is precisely for these problems with large m that the fast pairwise grouping heuristic is most needed.

We have developed a third pairwise grouping heuristic to remedy this discrepancy. This heuristic (we may call it the general pairwise grouping heuristic) is a combination of both the pairwise grouping heuristic and the fast pairwise grouping heuristic. The general pairwise grouping heuristic starts with the trivial partition, and attempts pairwise grouping in the same way as the fast pairwise grouping heuristic does, but only for a limited number of steps, say k_1 steps. (The fast pairwise grouping heuristic iterated for an unbounded number of steps until it could not produce an improvement to the last partition found.) The partition that results after k_1 steps is then taken and the fast

pairwise grouping heuristic is then reapplied to this partition but only for k_2 steps. This process is repeated (each time for a limited number of steps) until at some point the fast pairwise grouping heuristic produces a partition that it cannot improve upon. Everytime the fast pairwise grouping heuristic is reapplied, all pairwise attractivity measures of blocks are recomputed; even pairs of blocks that did not participate in a grouping since the last application of the fast pairwise grouping heuristic will have their attractivity measures recomputed. In this manner, if $k_1, k_2, \ldots$ are chosen such that they are comparatively smaller than the number of steps the pairwise grouping heuristic is expected to iterate if applied to the same problem, then discrepancies like the one encountered in the above problem will most likely be eliminated. $k_1, k_2, \ldots$ are most effectively chosen if $k_1 \geq k_2 \geq \ldots \geq 1$, because as the process of pairwise grouping approaches the optimal partition, pairwise block attractivity measures become smaller, and there is a greater chance that a pair of blocks may be attractive in one step but unattractive in another step. For example in the problem with $m = 30$, we chose $k_1 = 6$, $k_2 = 4$, $k_3 = 2$, and $k_4 = k_5 = \ldots = 1$. The reason that the general pairwise grouping heuristic is a combination of the two other pairwise grouping heuristics is that if $k_1 = k_2 = \ldots = 1$, then the general pairwise grouping heuristic will be the same as the pairwise grouping heuristic, while if k_1 is taken to be unbounded, then the general pairwise grouping heuristic will become the fast pairwise grouping heuristic.

We have tried the general pairwise grouping heuristic on all of the six problems with $m = 30$. It always found the same partition as the partition that the pairwise grouping heuristic found, and on the average the general pairwise grouping heuristic required the evaluation of 1168 partitions and took 1596 seconds of processing time to

accomplish that. This compares favorably with the pairwise grouping heuristic where the respective figures are 3984 partitions and 5244 seconds. The processing time depends on how the parameters $k_1, k_2, k_3, \ldots$ are selected. The larger that they are chosen to be, the faster the heuristic will be, but the smaller the probability that the partition found will coincide with the partition found by the pairwise grouping heuristic.

In the next chapter we will describe in detail one of the experiments that we have performed. We will provide the database parameters, the usage pattern parameters, and we will show the sequence of partitions found by the pairwise grouping heuristics. The problem we have chosen to elaborate in Chapter 6 is the one problem with $m = 8$ for which the pairwise grouping heuristic did not find the optimal partition.

CHAPTER 6

AN EXAMPLE OF ATTRIBUTE PARTITIONING

We have presented the combination of the fast pairwise grouping heuristic and the single attribute degrouping-regrouping heuristic as the main heuristic technique of our attribute partitioning system. We have also presented a number of other attribute partitioning heuristics. In Section 7 of Chapter 5 we reported on the results of a program of experimentation with the pairwise grouping heuristic, the fast pairwise grouping heuristic, and the single attribute degrouping-regrouping heuristic. We applied these three heuristics to a number of usage pattern histories in the context of different database environments. We also performed a number of experiments on the other attribute partitioning heuristics, and we reported the overall results of the experiments in the previous chapter. In summary, none of the other heuristics performed as well as the fast pairwise grouping heuristic; the fast pairwise grouping heuristic consistently produced the same partition as the pairwise grouping heuristic, while requiring only a fraction of its time. In all the cases that we tested, the fast pairwise grouping heuristic consistently produced either the optimal partition (as determined by exhaustive enumeration) or else a near optimal partition that differed from the optimal partition by less than one percent. In those cases where the resultant partition of the fast pairwise grouping heuristic was nonoptimal, by using the single attribute degrouping-regrouping heuristic to improve upon the resultant partition, we were able to obtain the optimal partition in at most three steps.

1. The Statement and Solution of the Problem

In order that the reader develop a feel for the form and magnitude of the partitioning problems that we have considered and the heuristics used to search for their solutions, we present an example of an attribute partitioning problem in this section along with the solutions obtained by applying a number of the attribute partitioning heuristics. We have included only 8 attributes in the relation of this example (B(8) = 4140), so that we may arrive at the optimal partition by exhaustive enumeration of all possibilities. The attribute partitioning problem considered here is typical of the problems we have solved in our work. It is also an example of a problem where the single attribute degrouping-regrouping heuristic was relevant.

Figure 1 shows the database parameters: m, n, A, S, l_i, and s_i. Figure 2 shows the usage pattern query types as described in the query type table. The query type table contains the frequency of each query type, the connectivity of the predicates (conjunction or disjunction) in the selection component, the attributes in the selection component, and the attributes in the projection component. The joint selectivity of the selection component (although not actually included in the query type table) is also depicted.

The results of applying the heuristics and the exhaustive enumeration procedure on this particular example are shown in Figures 3-6. The total cost of processing the set of queries, when the file is unpartitioned (i.e., the one-file partition), is 1881286 pages (Figure 3). This processing cost was calculated by the file cost estimator in the manner described in Section 4.4. Figure 4 shows the partition that results from applying the pairwise grouping heuristic; the fast pairwise grouping heuristic produced an identical result. Both heuristics

m = 8	The number of attributes in the relation.
n = 100,000	The number of tuples in the relation.
A = {1, 2, 3, 4, 5, 6, 7, 8}	The attributes of the relation.
{1, 3}	The indexed attributes
S = 1024 words	The system page size.
7693 pages	The number of pages in the relation

Attribute lengths and selectivities:

Attribute i	Attribute length l_i	Attribute selectivity s_i
1	2	10^{-3}
2	2	.001
3	6	.01
4	6	.1
5	10	.005
6	10	$2 * 10^{-4}$
7	20	$5 * 10^{-5}$
8	20	$8 * 10^{-3}$

Figure 1 Database parameters.

|Q| = 31 Number of query types.

Query types and frequencies:

Frequency	Predicate	Selection component	Projection component	Joint selectivity
200	conj.	[1]	[2 3 4 5 6 7 8]	1.0E-5
50	conj.	[2]	[4 5]	.001
50	conj.	[2 3 4]	[6 7]	1.0E-5
45	conj.	[2]	[1 3 4 5 6 7 8]	.001
25	conj.	[2]	[3 7]	.001
10	conj.	[3 4]	[5 6]	.001
10	conj.	[3]	[1 4 5]	.01
10	conj.	[3 4]	[6 7 8]	.001
10	conj.	[1]	[7 8]	1.0E-5
8	conj.	[6]	[1 2]	2.0E-4
5	conj.	[2 3 4]	[1]	1.0E-5
5	conj.	[2 4]	[6]	1.0E-4
5	conj.	[1]	[7 8]	1.0E-5
4	conj.	[4 5]	[1 2 3]	5.0E-4
3	conj.	[4 5]	[7 8]	5.0E-4
1	conj.	[8]	[4 5 6]	8.0E-5
1	conj.	[6]	[4 5]	2.0E-4
1	conj.	[4]	[5 6]	.1
1	conj.	[4]	[7 8]	.1
35	disj.	[1 2]	[3 4 5 6 7 8]	1.01E-3
11	disj.	[1 3 8]	[4]	1.00891E-2
20	disj.	[2 6]	[3 7]	1.1998042E-3
8	disj.	[2 3]	[4 5]	1.0990001E-2
3	disj.	[5 6 7]	[7 8]	5.2487477E-3
3	disj.	[6 7]	[1 4 5]	2.4999678E-4
2	disj.	[3 4]	[6 7 8]	.10899998
2	disj.	[6 7 8]	[1 2]	3.2997131E-4
2	disj.	[4 5]	[4 6]	.10449998
2	disj.	[1 2 6]	[1 2 6]	1.2097954E-3
1	disj.	[4 5]	[1]	.10449998
1	disj.	[1 2]	[1 2 6]	1.0099932E-3

Figure 2 Query type table.

iterated for 4 steps.

All the partitioning heuristics have been programmed in the programming language MDL [26]. The compiled versions of the pairwise grouping heuristic and of the fast pairwise grouping heuristic resulted in the partition P^3 in 22 seconds and 13 seconds respectively, and required the file cost estimation of 74 and 43 partitions respectively.

As it turns out, P^3, although near optimal, is not optimal and differs from the optimal partition by a negligible amount. The exhaustive enumeration procedure (also programmed in MDL), found the optimal partition P^4 of Figure 5 after trying all 4140 possible partitions. The exhaustive enumeration procedure required 2305 seconds of CPU time to generate all possible partitions and to cost evaluate them in order to arrive at the optimal partition; this is 2.3 orders of magnitude slower than the fast pairwise grouping heuristic.

Comparing P^3 of Figure 4 and P^4 of Figure 5, we see that in the near optimal partition P^3, attribute 3 is grouped with attribute 7, while in the optimal partition P^4, attribute 3 is grouped with attribute 8. Hence when the single attribute degrouping-regrouping heuristic was tried on P^3 (Figure 5), it produced the optimal partition P^4 in two steps and after the evaluation of 24 partitions. (The second step being the test to see if the heuristic could improve upon P^4.) As may be observed from partitions P^3 and P^4, none of the other degrouping heuristics applied to P^3 could have resulted in

C({{1, 2, 3, 4, 5, 6, 7, 8}}) = 1881286 pages

Figure 8. One-file partition performance cost.

Step	Partition	Partition cost in pages
0	P^0 = {{1}, {2}, {3}, {4}, {5}, {6}, {7}, {8}}	$C(P^0)$ = 286004
1	P^1 = {{1}, {2}, {3}, {4, 5}, {6}, {7}, {8}}	$C(P^1)$ = 277870
2	P^2 = {{1}, {2}, {3, 7}, {4, 5}, {6}, {8}}	$C(P^2)$ = 272447
3	P^3 = {{1, 4, 5}, {2}, {3, 7}, {6}, {8}}	$C(P^3)$ = 271840
4	no improvement	

Figure 4 The results of the pairwise grouping heuristic and the fast pairwise grouping heuristic.

Step	Partition	Partition cost in pages
0	P^3 = {{1, 4, 5}, {2}, {3, 7}, {6}, {8}}	$C(P^3)$ = 271840
1	P^4 = {{1, 4, 5}, {2}, {3, 8}, {6}, {7}}	$C(P^4)$ = 271814
2	no improvement	

Figure 5 The result of the single attribute degrouping-regrouping heuristic. P^4 is the optimal partition found by exhaustive enumeration.

P^4.

The single attribute degrouping heuristic, the double attribute grouping heuristic, and the double attribute degrouping-regrouping heuristic all produced partitions with higher performance costs than P^3.

We also tried other grouping heuristics starting from P^0. The triplewise grouping

heuristic resulted in a partition far less optimal than P^3. The two most attractive pairwise grouping heuristic found the optimal partition P^4, (although it has not always been the case that this heuristic would find the optimal partition). Although the k-partition pairwise grouping heuristic with $k = 2$ was able to find the optimal partition P^4, it found the optimal partition after cost evaluating 120 partitions, considerably more than the 67 partition evaluations required by the combination of the fast pairwise grouping heuristic and the single attribute degrouping-regrouping heuristic. For files with a larger number of attributes, the disparity in search cost (between these two heuristics) further increases.

The single attribute ungrouping heuristic was able to produce the optimal partition P^4 when started from the one-file partition. The converging sequence is shown in Figure 6. This is in accord with our good experience with the single attribute ungrouping heuristic.

Step	Partition	Partition cost in pages
0	$P^0 = \{\{1, 2, 3, 4, 5, 6, 7, 8\}\}$	$C(P^0) = 1881286$
1	$P^1 = \{\{1, 3, 4, 5, 6, 7, 8\}, \{2\}\}$	$C(P^1) = 602691$
2	$P^2 = \{\{1, 3, 4, 5, 7, 8\}, \{2\}, \{6\}\}$	$C(P^2) = 370471$
3	$P^3 = \{\{1, 3, 4, 5, 7\}, \{2\}, \{6\}, \{8\}\}$	$C(P^3) = 306233$
4	$P^4 = \{\{1, 3, 4, 5\}, \{2\}, \{6\}, \{7\}, \{8\}\}$	$C(P^4) = 283959$
5	$P^5 = \{\{1, 4, 5\}, \{2\}, \{3, 8\}, \{6\}, \{7\}\}$	$C(P^5) = 271814$
6	no improvement	

Figure 6 The result of the single attribute ungrouping heuristic.

The single attribute ungrouping heuristic required a total of 32 seconds of CPU time and terminated after considering 107 partitions.

2. The Efficacy of the Trivial Partition

In the example presented above, we see that the trivial partition P^0 in Figure 4 has a surprisingly low performance cost when compared with the one-file partition of Figure 3. In fact, the trivial partition in most examples turns out to be relatively near optimal, and for some combination of database usage pattern and database parameters, the performance cost of the trivial partition can be less than 30% percent of the performance cost of the one-file partition. In the course of our experimentation, it was also observed that the relative optimality of the trivial partition increased as the number of the attributes in the relation increased. We offer the following explanations for the relative optimality of the trivial partition:

(a) Most queries access only a few of the attributes. For every query that accesses a group of attributes, there are usually other queries that access a subset of this group of attributes (plus possibly other attributes) and hence cause the group to break apart (i.e., the subset of attributes becomes less attractive to the rest of the group). For example, if one query accesses attributes 1 and 2 and another query accesses attributes 2 and 3, and both have the same frequency, then (depending on the selectivities of the queries) it is most likely that a trivial partition of the attributes 1, 2, and 3 is more cost effective than a one-file or a two-subfile partition of them.

(b) In order that two attributes be grouped together in the same block, they have to be accessed in the same query. The more selective the query (i.e. the smaller the number of tuples that satisfy the query's predicate), the greater the advantage of grouping the attributes. If the query is very selective, grouping the attributes can reduce the number of page accesses to half the page accesses that would be encountered if the attributes were not grouped; therefore the attributes used in a selective query are very attractive to one another. Conversely, if a query is rather unselective, the relative advantage gained by grouping its attributes is slight. For example, if a query is so unselective that it requires all the pages of the subfile, then grouping all the attributes in one subfile will incur no more page accesses than separating them into several subfiles (which include no other attributes). Therefore, although grouping attributes becomes more desirable for selective queries, the unselective queries reduce the overall attractivity among attributes which has been induced by the selective queries. On the other hand, selective queries incur fewer page accesses than unselective queries, and the performance cost of a partition is determined to a greater extent by unselective queries. Thus for a usage pattern where selective and unselective queries are represented with the same frequency, the optimal partition will be close to the trivial partition in order to reflect the dominance of the unselective queries.

(c) An attribute that is resolved by sequential searching or by linking from a TID list which contains a large number of TIDs (when compared to the total number of tuples in the relation) will manifest a smaller attractivity to the other attributes of the file. In contrast, an attribute that is indexed will have a relatively higher attractivity measure to

the other attributes of the file. This is because when an attribute is resolved by sequential searching or by linking from a large TID list, all the values for that attribute will have to be retrieved in their entirety. Any attribute that may co-exist with this attribute in the same subfile will also have to be retrieved in its entirety, even if this attribute is not requested in the query that requests the unindexed attribute. Hence (as we have noted in Section 5.7), providing indices on attributes will contribute to the clustering of the attributes. In the experiments we have conducted (and also in most real databases), only a few of the file's attributes were indexed and most attributes did not have an index. Therefore for such cases, the trivial partition will be relatively optimal, since the attributes that are not indexed will be unattractive or show little attractivity to the rest of the attributes of the file, and the optimal partition will be composed of a large number of subfiles, each subfile containing only a few attributes.

The merit of the trivial partition has not been overlooked in practical database design. A variant of the trivial partition in a non-flat file implementation of a database is a file organization where the main file consists of records of pointers, in which each pointer points to the actual attribute value stored in a secondary data area [25]. The secondary data area is separate from the main file, and values of different attributes are stored separately. In this manner, the attributes are separated from one another into their own exclusive areas in a way that resembles a trivial partition. Accessing one attribute will therefore not bring in any values of other attributes (except for the pointers which are relatively short), but will cause one extra page access since the pointer has to be followed to the actual data. In such file organizations, data compaction techniques (like eliminating redundant copies of the same

attribute value) becomes possible, and are often employed to reduce storage (and also access) requirements (but at the expense of computing time).

CHAPTER 7

REMARKS AND FUTURE DIRECTIONS

In this report we have outlined a self-adaptive relational database management system that performs attribute partitioning for single relation databases. We have also developed a number of computationally feasible attribute partitioning heuristics that select a near optimal partition for the attributes, with the goal of optimizing on the paging performance of the database. In the next section, we provide suggestions for extending the underlying environment in order to solve more realistic attribute partitioning problems, without introducing the need for revamping the attribute partition heuristics. Thereafter, we conclude this report with suggestions for extending the scope of attribute partitioning to wider problems, along with a discussion of the relationship between database attribute partitioning and other physical database design issues.

1. Extensions to the Attribute Partitioning System

There are numerous issues and parameters that have to be considered when optimizing the physical design of a database in a complex environment. In this report we have addressed the attribute partitioning problem in the context of a model of the database management system that incorporates a number of those parameters that we feel are more important and whose consideration is required in order to have a meaningful model of practical database usage. Two of these parameters are attribute selectivity and the blocking

of tuples into discrete pages. However, it is possible to incorporate into our model of the database management system a number of other parameters without increasing the complexity of the attribute partitioning problem, and without introducing the need for revamping the heuristics we have shown to be appropriate in our current model. In the extensions to our current model suggested below, we believe that the pairwise grouping heuristic will still be the most viable heuristic and will retain its near optimality as demonstrated in the course of our work. Our belief is based upon the result of our program of experimentation with the heuristic, showing that the partition produced by the pairwise grouping heuristic is consistently within 1% of the optimal partition (for problems where the optimal partition could not be identified, within 1% of the best partition found for that problem). However, it is possible that in an extended model, the attractivity measure of a pair of blocks will not be the same from one step of the pairwise grouping heuristic to the next step (i.e. violating condition 5.5.1), thus precluding the application of the fast pairwise grouping heuristic.

One parameter that would be desirable to include in our model is the overhead cost of accessing a subfile. When a query is made to the database, there is usually a degree of overhead associated with accessing each subfile. The overhead may be incurred in opening and closing the subfile, initializing the page map table for the subfile, or allocating buffer areas in primary memory for the subfile. Overhead charges can be incorporated in our model by having the file cost estimator add a fixed overhead cost to the total cost figure whenever a subfile is accessed by the method of a query.

One other way we may extend our current model is to differentiate between the cost

of accessing different subfiles and to differentiate between the storage cost for different subfiles. It is possible to store subfiles on a variety of storage devices with differing storage and access costs, in order to further reduce the database performance cost. For example, a subfile that is not frequently accessed and that requires a significant amount of storage may be stored on a secondary storage device, other than a disk, that has higher access cost but lower storage cost. Differentiation between the costs of subfiles has been the main concern of some previous studies that have viewed the attribute partitioning problem in terms of allocating attributes to primary and secondary subfiles (Bonner [4], Hoffer [18], Eisner and Severance [14], and March and Severance [23]). We may incorporate differentiated subfiles in our model by making the file cost estimator multiply the number of page accesses to a subfile by a factor that reflects the relative cost of accessing the pages of the subfile with respect to the other subfiles. Differentiated storage costs for subfiles may also be included in our model by adding up the storage costs for each subfile depending on the kind of device the subfile is stored on during the time interval between repartitioning points.

It would be desirable to have the system automatically and optimally determine the repartitioning points. Computing the repartitioning points should be based upon the consideration that too frequent invocation of the partitioning heuristics will result in expending a significant amount of computational resources on the search for a marginally better attribute partition, while infrequent invocation of the heuristics may result in degraded database performance in the intervening time between repartitioning points. Shneiderman [33] and Yao et al. [38] have investigated the problem of optimally determining reorganization points for the purpose of eliminating overflows in files that are due to tuple

insertion and deletion. They also present a number of techniques for this purpose. We feel that their techniques could be adapted to the problem of determining optimal repartitioning points.

2. Directions for Future Research

Even though our investigation of attribute partitioning is in many respects more comprehensive than previous studies, we have only considered the attribute partitioning problem within the context of a single relation database environment, and where the relation is accessed through a restricted interface with limited capabilities for the selection of data. To fully realize the flexibility of a relational database, it is necessary to consider a multi-relation environment together with a high-level nonprocedural language interface that permits queries with arbitrary join operations between relations in their qualification parts. Also, it is necessary to consider queries that have arbitrary boolean expressions (ie. any combination of conjunctions and disjunctions) in their qualification part, and in which the predicates of the boolean expression contain other comparison operators besides equality conditions. (Arbitrary boolean expressions in queries have been disallowed in our model mainly because of the complications that arise in the evaluation of such queries for a partitioned database. Also, the consideration of comparison operators other than equality conditions makes the problem of estimating the number of tuples that satisfy a query considerably more complex, though not infeasible.) To our knowledge, the attribute partitioning problem for a multi-relation database with generalized queries has not yet been addressed.

We have assumed a flat file physical implementation of a relation in our attribute

partitioning system. In most practical databases, nonflat file implementation of data is assumed for two reasons. 1- For the purpose of data compaction, in order to reduce the amount of storage required to store the data. Data compaction may be accomplished by replacing attribute values with pointers that point to a common but smaller pool of the actual attribute values (data encoding), by using pointers that point to the end of a variable length attribute value (so that the maximum storage space does not have to be allocated for each of the variable length attributes), or by "factoring out" the same attribute value from any number of tuples that contain the attribute value, in order to eliminate its repetition in more than one tuple (as in hierarchical organizations). 2- For the purpose of access enhancement. Access enhancement in a nonflat implementation may be accomplished by the elimination of join operations (i.e. searches) on subfiles by having tuples that satisfy a frequent join operation explicitly linked to one another by pointers. We believe that the development of attribute partitioning heuristics for nonflat file implementations will not be an easy task; this difficulty is compounded by the complexity of query evaluation in such file organizations and by the problems of deriving an accurate cost model of such a database management environment. Babad [2] and Benner [4] have addressed a limited form of the attribute partitioning problem in a nonflat file implementation of a relation. Specifically, they consider the problem of attribute partitioning for file organizations that allow attribute values with variable lengths, and where not all subtuples are of equal length. Schkolnick [31] considers a hierarchical file organization where each node in the tree is an attribute. Page access is minimized by partitioning the nodes of the hierarchic file according to the frequency of attribute requests by queries and according to the position of the attribute within the hierarchy.

We have confined our attention to problems of attribute partitioning. Attribute clustering, as the logical extension of attribute partitioning, can result in considerable access (and storage) cost reduction over simple partitioning. As an example where attribute clustering may result in access cost reduction, consider attributes a_1, a_2, and a_3, where a_1 is highly attractive to both a_2 and a_3, but where a_2 and a_3 are highly unattractive. In such a situation it is desirable to redundantly store a_1 in two subfiles: in one subfile with a_2 and in the other subfile with a_3. Furthermore, in some cases, attribute clustering can paradoxically result in the reduction of storage costs. If a set of attributes of a relation is reproduced in a subfile of the attribute cluster, such that all the other attributes of the subfile are functionally dependent on these reproduced attributes, then it is possible to replace linking as an access path to subtuples of the subfile by an equi-join on the reproduced set of attributes. In other words, for a given subtuple, in order to locate its co-subtuple in a subfile, the equi-join of subtuples in the subfile is taken with the given subtuple on the set of reproduced attributes. Since the set of reproduced attributes constitutes a key for the subfile, the result of the equi-join is a single subtuple which is the co-subtuple of the given subtuple. The consequence of eliminating the links is that the subfile containing the reproduced copy of the attributes may be compacted by eliminating all identical subtuples. In certain cases this will result in considerable recovery of spare storage as all the subtuples of the subfile become unique. But there are more direct techniques for saving storage (such as data encoding) that are better for this purpose than attribute clustering. Therefore attribute clustering should be primarily considered for the purpose of access cost minimization. Osman [29] considers a special case of attribute clustering in a single relation database environment where the primary key of the relation is reproduced in each of the subfiles. The original relation may

then be reconstructed from the equi-join of the collection of the subfiles on the primary key. This model may be construed as a variant of attribute partitioning where the linking access path is replaced by the equi-join operation, since by reproducing the primary key all subtuples become unique, each subfile has n subtuples, and a subtuple has a unique co-subtuple in each of the other subfiles. Hoffer [18] also addresses the problem of attribute clustering in a flat file implementation of a relation and constructs an integer programming formulation of the problem, together with a branch and bound algorithm for its solution.

A problem related to attribute partitioning and attribute clustering is their dual: tuple partitioning (or horizontal partitioning, where the file is partitioned horizontally by tuples.) It is well known that tuples are not requested by queries with uniform frequency; in actual applications it has been frequently observed that the tuple request distribution follows the "80-20" rule of thumb [17]. This rule states that 80% of the queries deal with the most active 20% of a file. Therefore considerable access enhancement may be accomplished by clustering the frequently accessed tuples together, separate from the infrequently accessed tuples. In its simplest form, tuple partitioning is accomplished in a flat file implementation by grouping tuples that are accessed together into horizontal subfiles. The reason such a subfile is called a horizontal subfile is because the table representing the file is being partitioned horizontally by its rows. (Similarly, a subfile of an attribute partition may be viewed as a vertical subfile.) In tuple partitioning, in order that the access cost of two similarly accessed tuples be reduced, the tuples have to be placed in the same page. Tuple partitioning may thus be viewed as shuffling the tuples among the pages of the file, placing similarly accessed tuples in the same page, so that total access cost is minimized. We feel that the attribute

partitioning heuristics developed in this report will be computationally infeasible if ever applied to the tuple partitioning problem because of the sheer number of tuples involved. In practical databases, the number of tuples in the database is frequently as large as 10^5. For the same reason, it appears rather unlikely that a stepwise minimization heuristic that considers each individual tuple can be developed and successfully applied to the tuple partitioning problem. Another source of difficulty in the tuple partitioning problem is that a subfile in a tuple partition is limited in capacity (i.e. limited to a page). The heuristics we have developed do not take such constraints into consideration.

Knuth [22] and Rivest [30] each describe a heuristic for clustering tuples according to their access frequencies. In the heuristic of Knuth, each tuple that is retrieved by a query is relocated to the top of the file such that the tuple will become the first tuple to be searched by the next query. In the heuristic of Rivest, a tuple that is accessed is exchanged with the immediately preceding tuple. Both heuristics are rather limited in scope since they assume no blocking of tuples into pages and they assume that the only way that the file is searched is by sequential search. The heuristics also have the drawback that they are very sensitive to the order the queries are made to the database, and the performance cost of the tuple partitioned database can drastically change if the queries arrive at a slightly different order.

We believe that the tuple partitioning problem should be solved using cluster analysis techniques that consider both the attribute values in the tuple and the occurence of attribute values in queries. Statistics gathered for the purpose of tuple partitioning must record not only the attributes requested by the queries made to the database, but also the attribute values in the equality condition predicates of the query, so that similarly accessed tuples may be identified.

A slight generalization of the tuple partitioning problem is to the case where tuples are stored in pages that need not be completely filled, but may be partly empty. This variation of tuple partitioning is even more difficult to solve, but may result in enhanced access performance at the cost of an increase in storage requirement. Tuple partitioning may be further generalized into tuple clustering, where tuples may be redundantly stored in several pages, in order to reduce page accesses (while increasing storage requirements).

A further generalization of the partitioning problem is the hybrid clustering (partitioning) problem where attribute clustering (partitioning) and tuple clustering (partitioning) are carried out simultaneously. In this problem, a file is partitioned both by its attributes and by its tuples such that each subfile has a subset of the attributes and a subset of the tuples. The subfiles of a hybrid partition need not all be of the same size either in the number of attributes or in the number of tuples. One way to picture a flat file partitioned in a hybrid manner is as a composition of rectangular mosaics, with varying lengths and widths, placed adjacent to one another such that the whole file is covered. The hybrid data clustering (partitioning) problem is much larger than either the attribute clustering or the tuple clustering problems. For this reason, its solution requires more powerful heuristics than any we have considered. A computationally feasible, yet not necessarily optimal or near optimal, approach would be to perform attribute clustering (partitioning) and tuple clustering (partitioning) alternately, in order to reach a hybrid cluster that has a locally minimum performance cost.

Another direction of extending attribute partitioning is to consider attribute partitioning simultaneously with selecting other file access structures, where the choice of the

access structure depends on the choice of the attribute partition, and vice versa. For example, we may consider choosing indices for the attributes of a partitioned file. In our current model, we have assumed the set of attributes that are indexed to be predetermined, and consequently the file is partitioned based on a fixed set of indices. If the restriction of fixed indices is removed, the overall performance of the database management system might improve when attribute partitioning and index selection are considered simultaneously. One plausible strategy here is to alternately perform attribute partitioning and index selection in a stepwise minimization fashion. The heuristic presented by Chan [7] for the optimal selection of indices, is particularly useful for such a strategy.

REFERENCES

[1] Alsberg P. A., "Space and Time Savings through Large Data Base Compression and Dynamic Restructuring", Proceedings of the IEEE, Vol. 63, No. 8, August 1975.

[2] Babad, J. M., "A Record and File Partitioning Model", Communications of the ACM, Vol. 20, No. 1, January 1977.

[3] Bayer, R., and McCreight, E., "Organization and Maintenance of Large Ordered Indexes", Acta Informatica, Vol. 1, Fasc. 3, 1972.

[4] Benner F. H., "On Designing Generalized File Records for Management Information Systems", Proceedings of the Fall Joint Computer Conference, 1967.

[5] Belford, G. G., "Dynamic Data Clustering and Partitioning", CAC #162, Center for Advanced Computation, Research in Network Data Management and Resource Sharing, University of Illinois at Urbana-Champaign, May 1975.

[6] Blasgen, M. W., and Eswaran, K. P., "On the Evaluation of Queries in a Relational Data Base System", Computer Science Department, IBM Research Laboratory, San Jose, Calif., 1976.

[7] Chan, A. Y., "Index Selection in a Self-adaptive Relational Data Base Management System", S.M. Dissertation, Department of Electrical Engineering and Computer Science, MIT, Sept. 1976.

[8] Chan, A., and Niamir, B., "Estimating Record Accessing Cost in Blocked Database Organizations", Forthcoming Memorandum.

[9] Codd, E. F., "A Relational Model of Data for Large Shared Data Banks", Communications of the ACM Vol. 13, No. 6, June 1970.

[10] Codd, E. F., "Further Normalization of the Data Base Relational Model", Courant Computer Science Symposia 6, New York NY, 24-25 May 1971, in Data Base Systems, Prentice Hall, 1971.

[11] Day, R. H., "An Optimal Extracting from a Multiple File Data Storage System: An Application of Integer Programming," Operations Research, 13, 3 (May 1956) 482-494.

[12] Dearnley, P., "Model of a Self-organizing Data Management System", Computer Journal Vol. 17, No. 1, Feb. 1974.

[13] Delobel, C., and Casey, R. G., "Decomposition of a Data Base and the Theory of Boolean Switching Functions", IBM Journal of Research and Development, September 1973.

[14] Eisner, M. J., and Severance, D. G., "Mathematical Techniques for Efficient Record Segmentation in Large Shared Databases", Journal of the Association for Computing Machinery, Vol. 23, No. 4, October 1976, pp. 619-635.

[15] Hammer, M. M., and Chan, A. Y., "Acquisition and Utilization of Access Patterns in Relational Data Base Implementation", Pattern Recognition and Artificial Intelligence (editor Chen, C. H.), Academic Press, 1976.

[16] Hammer, M. M., "Self-adaptive Automatic Data Base Design", Proceedings of National Computer Conference, Dallas, Texas, June 1977.

[17] Heising, W. P., "Note on Random Addressing Techniques", IBM Systems Journal, Volume 2, June 1963.

[18] Hoffer, J. A., "A Clustering Approach to the Generation of Subfiles for the Design of a Computer Data Base", Ph. D. Dissertation, Department of Operations Research, Cornell University (January 1975) 276pp.

[19] Hoffer, J. A., and Severance, D. G., "The Use of Cluster Analysis in Physical Data Base Design", Proceedings of International Conference on Very Large Data Bases, Framingham MA, Sept. 1975.

[20] Kennedy, S. R., "The Use of Access Frequencies in Data Base Organization", Ph. D. Dissertation, The Wharton School, University of Pennsylvania (May 1973), 155pp.

[21] Kennedy, S. R., "A File Partitioning Model", California Institute of Technology Information Science Technical Report 2, May 1972.

[22] Knuth, D., "Sorting and Searching", The Art of Computer Programming, Vol. 3, Addison-Wesley, 1973.

[23] March, S. T. and Severance D. G., "The Determination of Efficient Record Segmentation and Blocking Factors for Shared Data Files", ACM Transactions on Database Systems, Vol. 2, No. 3, Sept. 1977.

[24] McCormick, W. T., Jr., and Schweitzer, P. J., and White, T. W., "Problem Decomposition and Data Reorganization by a Clustering Technique," Operations Research, 20, 4 (July 1972) 741-751.

[25] McLeod, D., Meldman, M., Pellicore, R., and Squire, M., "RISS: A Relational Data Base Management System for Minicomputers", Van Nostrand Reinhold, 1978 (to appear).

[26] MDL Primer and Manual (for versions 54 and 104), Galley, S. W., and Pfister, G., MIT Laboratory for Computer Science, May 1977.

[27] Moser, L., and Wyman, M., "An Asymptotic formula for the Bell numbers", Transactions of the Royal Society of Canada, Section III, Vol. 49 (1955) pp. 49-54.

[28] Niamir, B., "Asymptotic Upper and Lower Bounds for the Bell Numbers", Data Base Systems Group, MIT Laboratory for Computer Science.

[29] Osman, I. M., "The Partitioning of a Data Base into Supplies Matching User's Queries", Computer Centre, University of Khartoum, Sudan.

[30] Rivest, R., "On Self-Organizing Sequential Search Heuristics", Communications of the ACM, Vol. 19, No. 2, February 1976.

[31] Schkolnick, M., "A Clustering Algorithm for Hierarchical Structures", ACM Transactions on Database Systems, Vol. 2, No. 1, March 1977.

[32] Seppälä, Y., "Definition of Extraction Files and Their Optimization by Zero-One Programming," BIT, 7, 3 (1967) 208-215.

[33] Shneiderman, B., "Optimum Data Base Reorganization Points", Communications of the ACM, Vol. 16, No. 6, June 1973.

[34] Small, D., "A Data Base for Application to the Research and Development of Advanced Navy Command and Control Systems", Naval Electronics Laboratory Center, San Diego, Calif., NELC J752, Feb. 1976.

[35] Stocker, P. M., and Dearnley, P. A., "Self-organizing Data Management Systems", Computer Journal 16, 2 (May 1973).

[36] Waters, S. J., "Hit Ratios", The Computer Journal, Volume 19, Number 1, April 1976.

[37] Yao, S. B., "[illegible]", [illegible]

[38] Yao, S. B., Das, K. S., and [illegible], T. J., "A Dynamic Database [illegible]", [illegible], Vol. [illegible]

[39] Yue, P. C., and Wong, C. K., "On the [illegible] of a Partition Algorithm", [illegible]

[40] Yue, P. C., and Wong, C. K., "[illegible] in Secondary [illegible]", [illegible]

This empty page was substituted for a blank page in the original document.

CS-TR Scanning Project
Document Control Form

Date : 10/26/95

Report # LCS-TR-192

Each of the following should be identified by a checkmark:
Originating Department:

☐ Artificial Intellegence Laboratory (AI)
☒ Laboratory for Computer Science (LCS)

Document Type:

☒ Technical Report (TR) ☐ Technical Memo (TM)
☐ Other:_____

Document Information

Number of pages: 158(164-IMAGES)
Not to include DOD forms, printer intstructions, etc... original pages only.

Originals are:
☐ Single-sided or
☒ Double-sided

Intended to be printed as :
☐ Single-sided or
☒ Double-sided

Print type:
☐ Typewriter ☐ Offset Press ☐ Laser Print
☐ InkJet Printer ☒ Unknown ☐ Other:_____

Check each if included with document:

☐ DOD Form ☐ Funding Agent Form ☒ Cover Page
☒ Spine ☐ Printers Notes ☐ Photo negatives
☐ Other: _____

Page Data:

Blank Pages(by page number): 6, BLANK FOLLOWS LAST PAGE.

Photographs/Tonal Material (by page number):_____

Other (note description/page number):

Description : Page Number:
IMAGE MAP: (1-158) UNTITLED TITLE PAGE, 2-157, UNTITLED BLANK.
(159-164) SCANCONTROL, COVER, SPINE, TRGT'S (3).

Scanning Agent Signoff:

Date Received: 10/26/95 Date Scanned: 11/15/95 Date Returned: 11/16/95

Scanning Agent Signature: Michael W. Cook

Scanning Agent Identification Target

Scanning of this document was supported in part by the **Corporation for National Research Initiatives,** using funds from the **Advanced Research Projects Agency** of the **United states Government** under Grant: **MDA972-92-J1029.**

The scanning agent for this project was the **Document Services** department of the **M.I.T Libraries.** Technical support for this project was also provided by the **M.I.T. Laboratory for Computer Sciences.**

Scanned

Date: 11/15/1995

M.I.T. Libraries
Document Services