23

PARTITIONING A DATABASE TO PROVIDE SECURITY AGAINST STATISTICAL INFERENCE

© David Michael Glaser

A Thesis

in

The Faculty

of

Engineering and Computer Science

Presented in Partial Fulfillment of the Requirements
for the degree of Master of Computer Science at
Concordia University
Montreal, Quebec, Canada

February 1982

ABSTRACT

PARTITIONING A DATABASE TO PROVIDE SECURITY AGAINST STATISTICAL INFERENCE

David Michael Glaser

This thesis investigates partitioning as a means of preventing disclosure of confidential information in a statistical database. All records are placed into small disjoint groups. By providing statistics only about entire groups the data in an individual record cannot reliably be ascertained.

Two methods are presented for achieving the partitioning. One is a bottom-up technique which builds partitions from single records. The other is a top-down method which produces the partitions by successively splitting the entire database into small groups.

Analysis of the algorithms and experiments performed on a simulated database show that partitioning can provide effective control of statistical compromise at low cost, while producing accurate statistics.

To my wife Perle

and daughter Alison.

ACKNOWLEDGEMENTS

I would like to express my gratitude to Professor V.S. Alagar for his guidance and assistance in the completion of this thesis. Professor Alagar has taught me much, not only about computer science, statistics, and analysis, but of the rewards in research and the pursuit of knowledge.

I also wish to thank Beverley Abramovitz for a superb job of typing, and my colleagues and the secretaries in the Department of Computer Science at Concordia University for their help and suggestions.

I am grateful for the financial assistance provided by Professor Alagar and the Department of Computer Science.

Finally, many thanks to my wife Perle, whose support and confidence never wavered.

This work has been supported by funds from the National Research Council of Canada.

TABLE OF CONTENTS

CHAPTER page

CHAPTER page

CHAPTER ONE

INTRODUCTION

The use of computers to store personal information for research on social and economic issues is increasing rapidly. As moderately sized databases become more commonly available and very large databases become accessible to large institutions, the social impact of these systems must be considered. Problems of security, of safeguarding the privacy of the individuals whose files are stored in the computer and of controlling access to those files, are now problems for the computer scientist as well as social policy makers. There is growing concern that the confidentiality promised when individuals give information to institutions such as banks, hospitals, census bureaus and researchers cannot be guaranteed if that information is stored in on-line data banks. It is feared, somewhat justifiably, that storing records in computers increases the ways in which an unauthorized person can access that information by circumventing the modest security built into most systems.

This thesis addresses one particular database security problem. In a database in which any user can obtain statistical summaries about subgroups of its population, the important problem is that of protecting against inference of any individual's information. This is a particularly distressing security matter since there

is no question of unauthorized access. Any user should be able to obtain statistical summaries but should not be able to infer any individual data. If no restrictions are placed on the statistics to be released it is very easy to acquire information about anyone whose record is in the database. Certain obvious restrictions such as refusing to answer queries about very small groups make such inferences harder but still inference is relatively easy to perform. We present in this thesis methods for securing statistical databases which, by distorting the statistics to a small degree, renders information inferred about an individual completely unreliable.

1.1 The statistical inference problem

A database which is primarily designed to provide statistical summaries about subgroups in its population will be called a "statistical database." Medical information in hospitals, financial records kept by banks and sociological information collected by the Census Bureau are typical instances of statistical databases.

We can broadly describe two categories of users of a statistical database. The first has authorization to read, write and update data while the second can only request statistics about the data. Effective database security measures must prevent unauthorised users from tampering with the data or accessing confidential information not normally available to them. This problem has been widely studied and encryption schemes and other protection

mechanisms can enforce these safeguards [11].

It is the latter group of users which concerns us here. These users may request extensive statistics on subsets of the database, and may thus be able to fairly easily and completely legally discover individual information which was promised to be confidential.

This section explores the problem of inference in computerized statistical-reporting and research systems. We will show how individual information can be deduced and discuss the significance of the problem, not only to computer scientists but to social policy makers and the society as a whole.

1.1.1 Storing records on computers

Recent decades have seen a tremendous growth in the power and availability of computer systems. Even in the infancy of the computer industry people were concerned about the adverse effects of maintaining personal records in machines. The process was seen by some as dehumanizing and uncontrollable. The prospect of a large centralized data bank with dossiers about everyone and accessibility to a wide number of users was particularly frightening. Others claimed that the computer itself was harmless; it was the way people used it that was the problem.

There are in fact problems which arise with the use of computers to store records. These stem from the following sources:

- the ability to store large amounts of data;
- easy access to that data;
- computer technicians serving as record keepers;
- difficulty responding to the community.

The first major problem is a result of the increasing ability to store large amounts of data both in large systems and in mini-computers which are only now becoming economically feasible for small organizations to operate. Indeed, the cost of setting up an automated record system may compel a smaller organization to store more data than it really needs in order to more fully utilize the system. This is a dangerous policy since it not only threatens the privacy of the individual but may obstruct the access to more pertinent information. Not only is there a wealth of information but that data is, by the very nature of the machine, somewhat rigid. Most systems cannot handle unusual responses on input forms. Each question has a limited number of response alternatives and some subjects may not feel that any apply to them. It is much harder for a computer to deal with the category "other" as a response than a manual records system which may simply attach an explanation.

It is also easier to access data in a machine. This is not a matter of securing the system against unauthorized access. Security in that sense can be enforced as well in an automated system as a manual one, if not better. It is rather a problem of prohibiting authorized use for

unauthorized purposes. In the U.S. Department of Health, Education and Welfare's Report on Automated Personal Data Systems [27] the authors describe what they call "dragnet" behavior. By this they mean "any systematic screening of all members of a population in order to discover a few members with specified characteristics."[1] Clearly this is only feasible in a computerized filing system and is tempting because of the small amount of human labor involved.

Another threat posed by such a system is the ease with which one organization maintaining an extensive set of records can send information it has gathered from the system to other organizations. The controls on such a transfer and responsibility for ensuring the proper use of the information may not be at all clear. This would be of tremendous concern if the government kept a large centralized data bank which would be available to many different types of users. At present, however, such a system is not imminent.

The widespread use of automated record systems has resulted in a new class of record keepers who are, by trade, computer technicians. Their primary function is to facilitate the use of the system. They make no judgments on the validity of that use and indeed their contact with

---

[1] Records, Computers and the Rights of Citizens, by U.S. Department of HEW (n.p.: the Colonial Press, 1973).

the suppliers and users of the information in the system is minimal.

Finally, a computerized collection of records is somewhat immune to feedback from the community. Because of its highly technical nature, even when everyone concerned agrees to necessary changes in the system, it may be hard and take a long time to implement.

The problems cited above support the view that concern over automated record keeping is justified. We must not forget, however, that there is a considerable benefit to being able to store and easily access large amounts of information which would be impossible or extremely difficult and time-consuming to operate manually. Automated medical records, for example, which could be accessed by a hospital physician, family doctor or specialist would eliminate much duplication of work and could save a number of lives. We must work toward safeguarding the privacy of the individuals whose records are being kept so that we may take advantage of the increased facility for data processing afforded by the computer.

1.1.2 Confidentiality assurances

The right of privacy is one which applies to all automated personal record-keeping systems. Files in most systems contain a unique identifier which is used to access the record. It is easy to see that such files should be protected from misuse. Records gathered and kept solely for generating statistics are fundamentally different in

that they have no identifiers or those identifiers are hidden and only used internally. The threat to privacy in such a system is less obvious but is nevertheless very real.

We have recently been obligated by law to participate in the Canadian census. Someone from every dwelling filled out a questionnaire which assured that the information provided would be used "for statistical purposes only" even though someone's name was asked for at least once on the form. Even if one believes that the government has the best intentions and no one at Statistics Canada who sees the form will use that data to their advantage, how can we be certain that when they release the statistics our privacy is assured? The statistics compiled from the census are released to government agencies to facilitate policy decisions. Any researcher in need of data which has been gathered in the census can also request the appropriate statistics. If a record contains enough information to uniquely identify the subject, his confidentiality is in jeopardy and safeguards must be taken to prevent disclosure of any non-public data.

The census bureaus both here and in the United States, where the most extensive computerized census ever undertaken is now being compiled, were among the first to recognize that there was a statistical inference problem and to deal with it. We will discuss their methods and other proposed safeguards in the next chapter.

The H.E.W. Report on Automated Personal Data Systems recognized this problem but assumed that most statistical data could not be traced to individuals. They discovered, however, that "in many instances files used exclusively for statistical reporting and research do contain personally identifiable data, and that the data are often totally vulnerable to disclosure through legal process."[2] They proceed to recommend that before any transfer of individually identifiable data takes place the organization maintaining such records must specify the security requirements of the data and determine that those requirements will be carried out, unless the individuals have given prior consent to the transfer.[3] This seems to be a reasonable suggestion but it is questionable how well it can be enforced if made into law.

1.1.3 Ease of compromise

We have recognized the problem of statistical inference in automated record-keeping systems and its significance to anyone who participates in a statistical survey. There would not be cause for alarm, though, if it were difficult to perform such deductions on a statistical database.

---

[2] Records, Computers and the Rights of Citizens, by U.S. Department of HEW (n.p.: the Colonial Press, 1973), p.91.

[3] Records, Computers and the Rights of Citizens, by U.S. Department of HEW (n.p.: the Colonial Press, 1973), p.98.

Hoffmann and Miller [23] were the first to point out how, with enough generally public information about a person in the data bank, one can infer sensitive information. The following is an example of their reasoning. Suppose we wish to find out John Doe's salary and we know the following information about him: that he is a ~~labor~~ leader and that he is a delegate to the Republican Presidential Convention. We then ask a computerized data bank on the U.S. Presidential Conventions "how many people are in the data bank with the following properties: labor leader
Republican delegate?"

If the system responds with "one person" we have isolated John Doe's record. This actually was the case at the 1980 convention as was demonstrated by a CBS computer. We can now ask "How many people are in the data bank with the following properties: labor leader
Republican delegate
Salary over $50,000?"

If the answer is "one person" we know John Doe earns over $50,000. If the answer is zero we lower our estimate. In fact, any attribute about John Doe which is in the data bank can be determined by simply adding it as an extra condition.

The immediate response to this type of attack is to refuse to answer queries about very small groups. The problem is that the same principle can be applied to larger groups. Schlörer [30] devised a method of padding the response group so that its size will always lie in the restricted range. By essentially adding unrelated records

to the isolated record we can then proceed as before. Suppose we ask "How many people are in the data bank with the following properties: labor leader and Republican delegate, or female?" Let us say the answer is 751. We subsequently ask for the number of people with the following properties: labor leader and Republican delegate and convicted of a felony, or female. An answer of 751 means John Doe has been convicted of a felony whereas the answer 750 means that he has not been convicted. Schlörer called this technique "tracking" and he called the formula which defines the large group a "tracker". Denning [12,13] has extended Schlörer's work and shown that trackers are very easy to find in almost any database.

Trackers are such a powerful tool of inference that preventing disclosure by restricting the answerable queries becomes virtually impossible. Other methods are needed which control the response to the queries so that the user cannot be certain the record he is trying to isolate has been included in a given query response. One such method, which we have implemented, involves partitioning the database into small groups. Statistics are calculated on the basis of the groups rather than individual records. This technique is introduced at the end of this chapter and discussed in detail in Chapter Two. We now present a formal model of a statistical database and define compromise of the database.

1.2 Basic definitions

This section presents a model of a statistical database including the types of queries which can be asked of the system. The concept of compromise in such a database is also defined and the different levels of compromise are discussed.

1.2.1 Statistical database model

Our conception of a statistical database is that of a single relation over a fixed number of attributes. A record R has k attributes and is defined as a k-tuple $(r_1 \ldots r_k)$ where r_i, the value in the ith attribute comes from a domain Di, $1 \leq i \leq k$. If any one attribute uniquely identifies each record it is called a key, and its corresponding domain a key domain. All the records must have the same attributes and the same number of attributes. Some of these may be qualitative or descriptive attributes while others may be quantitative or data fields. For example, the qualitative attributes may be SEX, OCCUPATION and MARITAL STATUS with their corresponding domains (Male, Female), (Professor, Student, Doctor, Lawyer) and (Single, Married, Separated). In addition, there may be quantitative attributes such as SALARY and Previous Psychiatric Admissions. A sample database with these attributes is given in Figure 1.1.

Statistics are obtained through queries of the database. These queries consist of a predicate expression which specifies one or more values of one or more attributes

No.	Unique Identifier	Sex	Occu-pation	Marital Status	Salary ($k)	No. Previous Admissions
1	Adams	M	Doctor	Married	28	3
2	Baird	M	Student	Single	5	1
3	Carney	F	Lawyer	Separated	20	5
4	Davies	F	Doctor	Married	20	0
5	Eaton	M	Pro-fessor	Single	28	0
6	Finch	F	Student	Married	5	0
7	Gagnon	M	Lawyer	Married	50	2
8	Harris	F	Doctor	Single	50	0
9	Ibsen	F	Pro-fessor	Separated	15	2
10	Jones	M	Pro-fessor	Married	20	0
11	Kapp	M	Lawyer	Single	50	0
12	Lewis	M	Doctor	Separated	15	6

Figure 1.1 Sample Database for a Psychiatric Hospital

related by the boolean operators AND (•), OR (+) and NOT (-). The subset of records whose characteristics match this expression will be called the "query set".

The unique identifier, the patient's name in our sample database, may or may not be accessible to the user. If a query is allowed to specify one or more keys it is called a "key-based" query. An example of a key-based query of the sample database is: Average (Adams, Eaton, Kapp; salary) which requests the average salary of Adams, Eaton and Kapp. If, on the other hand, the unique identifier is hidden from the user only subsets of the attributes may be used for retrieval purposes. Queries of this type are called "attribute-specified" queries and their predicate expression is known as a "characteristic formula". Count (M • Student) is an example of such a query asking for the number of male students in the database.

The type of allowable query is determined by the user environment. Names on psychiatric records would not normally be released to researchers requiring statistical information. Key-based queries are generally used more in small databases where records are easily identifiable anyway. The inference controls which we will discuss in this thesis are all concerned with attribute-specified queries since most statistical databases do not give out individual information. If key-specified queries are allowed they must be in a highly restricted form, as we shall see later.

Queries can be further classified by the type of statistic asked for. A "count query" requests the number of records satisfying the query's characteristic formula. Clearly this type of query must be attribute-specified for a key-specified query already contains that information. "Sum" or "average" queries can be of either type and ask for the appropriate statistic about the records in the query set for some data field.

Our querying system permits only attribute-specified queries requesting counts, averages of data field values and relative frequencies. Let us denote (after Denning [10]) the characteristic formula as C and the query set of C as X_C. The general forms of the permissible queries are then:

$\text{Count}(C) = |X_C|$, the size of X_C.

$\text{Freq}(C) = \frac{|X_C|}{N}$ where N is the total number of records in the database.

$\text{AVG}(C,j) = \sum_{i \in X_C} V_{ij}/\text{Count}(C)$ where V_{ij} is the value of data field j in record i.

Some examples of queries on our sample database are:

Formal Query	Answer	Informal Statement
Count(M·Lawyer)	2	number of male lawyers
Count(F·(Doctor + Lawyer))	3	number of females who are doctors or lawyers
Freq(F·$\overline{\text{married}}$)	.25	relative frequency of unmarried females
Avg(F·Lawyer, admiss.)	5	average no. of admissions for all female lawyers

Formal Query	Answer	Informal Statement
Avg(Professor, salary)	21k	average salary of all professors

1.2.2 Compromise of the database

Compromise occurs when the user deduces confidential information from the responses to one or more queries. If the compromise reveals that an individual belongs to some category or has a particular data value it is called "positive compromise". It is also possible, however, to deduce that an individual does not belong to some category or does not have a particular data value. This is called "negative compromise".

If a user deduces any information (positive or negative) previously unknown to him about an individual he has achieved "partial local compromise" of the database. If he deduces all the information in the database about any individual record we say he has achieved "total local compromise". To illustrate this, let us refer again to the sample database of Figure 1.1. Assume a user knows that Ibsen is a female professor and is included in the psychiatric database. Also assume there are no restrictions on the queries which will be responded to as long as they conform to the types of permissible queries described above, and that the responses given are accurate. The user then asks the query: COUNT(F•prof) and receives the answer 1. He now knows that Ibsen is the only female professor in the database, but has not deduced any information about Ibsen. He can then pose the query:

COUNT(F·prof·married) for which the response is 0. The user has thus achieved a negative compromise: he has learned that Ibsen is not married. He has also effected a partial local compromise of the database. The user can then proceed to determine that she is separated in a similar fashion. Subsequently, suppose he asks: AVG(F·prof,salary) and gets the response 15k. He has now achieved a positive compromise of the database. Finally, if he queries: AVG(F·prof,admissions) and the system answers 2 he knows she has been admitted twice for psychiatric reasons. At this point he has achieved a total local compromise of Ibsen's record.

It is clear from this example that if a system responds truthfully to any allowable query, it can be easily locally compromised by anyone who knows that an individual's record is contained in the database and has enough preknowledge about that individual to isolate their record.

We can describe two stronger levels of compromise of a database. If any user obtains partial local compromise for a subset of records in the database we say there is "weak global compromise". A database is "strongly globally compromised" if every record is totally locally compromised, i.e., everything about the database has been deduced.

A response strategy is a method of responding to queries of a particular type in order to make compromise hard at any of the levels described above. The effectiveness

of a strategy should be measured in terms of its prevention of compromise at one of the levels and its ease of implementation. A database is strongly secure if partial local compromise is either impossible or extremely expensive to implement.

1.3 Contributions of this thesis

We have implemented a strategy to secure statistical databases based on partitioning the database into small disjoint groups. The statistics generated in response to queries of the system are calculated using group summaries instead of individual records. This way the individual data is never used to produce the statistics. If a record belongs to a particular query set, its small group is substituted instead. This method provides secure statistics by eliminating any possibility of isolating a record.

The accuracy of the statistics from a partitioned database depends in part on how well the partitions are formed. If all the records in a group are similar, the group will more accurately represent each particular record. Small groups are also desirable for the same reason.

There are two partitioning methods presented in this thesis. The first is a modification of the method proposed by Yu and Chin [33].

The technique involves mapping all the records in the database onto a k-dimensional matrix consisting of one cell for each possible k-tuple. The number of records in each cell is examined and cells are merged with their

neighbours until a group is formed which contains the predefined minimum amount of records needed to qualify as a partition. We call this method rectangular partitioning because the regions we form are rectangular parallelopipeds (for $k > 2$) of minimum size.

The second method performs a hierarchical splitting of the database. The attributes are ordered according to some criterion such as size of the domain. All the records are then partitioned into disjoint subgroups, one for each value in the first attribute. For example, suppose there are five values possible for the first attribute. All the records with the first value will form one subgroup, those with the second will form another and so on resulting in five disjoint subgroups. These are further split using the remaining attributes, until no more groups can be formed containing the minimum number of records.

Both techniques were tested by posing 300 randomly generated queries and calculating the relative error of statistics based on the partitions as opposed to the true statistics. Databases of size 100, 500 and 1000 were used as well as two different minima for partition sizes, namely three and five. In addition, tracker attacks on the system were made to test the actual security of the computations.

Our results show that the query responses were quite accurate, for the most part differing from the true values

by less than 10%. The individual values inferred from tracker attacks, on the other hand, were typically several hundred percent off from the actual values. A small number of inferred values were accurate to within 10% but it was impossible to predict which records or trackers would produce those results.

The results were similar for both partitioning techniques. The hierarchical method, however, provides a much simpler and faster way of obtaining the partitions.

The thesis is organized as follows:

A history of the problem presenting recent literature on the subject of statistical disclosure and the various forms of preventing it is given in the next chapter of this thesis. Chapter Three describes rectangular partitioning while Chapter Four presents the hierarchical technique. The details of the testing procedure and its results, including the tracker attacks, make up Chapter Five. A final summary and conclusions then form the last chapter.

CHAPTER TWO

HISTORY OF THE PROBLEM

2.1 Introduction

It was Hoffman and Miller's article [23] which first demonstrated the seriousness of the statistical inference problem. By describing a subject's characteristics in as much detail as possible, one can usually isolate their record and proceed to find out any unknown information about the subject. Refusing to answer queries with small query set sizes did not adequately protect the database as trackers provide an easy method of circumventing such controls [12,30].

A number of researchers then tried to quantify the compromisability of a database under certain specified conditions. Using combinatorial analysis, the pioneering study of Dobkin, Jones and Lipton [16] derives a formula for the least number of queries needed to compromise a database. Similar studies have also tested databases in this manner in an attempt to define conditions which assure security against inference [7,8,24,28,29,32]. These studies all use key-based queries so their results are not very useful, but they do give an insight into the problem.

As it became clear that restrictions on query set size and maximum overlap among queries were ineffective, other methods started to be investigated. One of the most

promising of these was perturbation of the data either by adding a random weight to data values, randomly rounding the data or giving ranges instead of exact answers. If carefully done, these methods can be effective [1,3,4,18]. Several studies have suggested threat-monitoring techniques as a supplement to perturbation [17,23,31]. By keeping a log of which queries are asked by which users it may be possible to detect unauthorized intrusion. Effective threat-monitoring is a complicated and almost impossible procedure, however, since collaboration between two or more users is very difficult to determine.

Recently there has been more emphasis on response set controls as a means of safeguarding statistical databases. These methods make it impossible for a user to know exactly which records are included in the query set. Notable techniques of this type are random sampling, which has been implemented by Denning [15] and partitioning which was described by Yu and Chin [33] and has been implemented by us as presented in this thesis.

This section describes the literature on the statistical inference problem. Emphasis is placed on studies using characteristic-specified queries and the methods which have been proposed to secure databases allowing this type of query. Several good survey papers have been published which provide a general description of the problem and a review of the literature to date [1,2,9,10,11].

2.2 Query set size controls

It is a simple matter to compromise a database which will answer truthfully any query regardless of how many records it involves. All one has to do is use enough characteristics describing an individual to uniquely identify him. These attributes can be added to the description, one at a time, until the system responds that there is only one such individual. Once that happens, any unknown characteristics can be included in the description. If the system indicates there is still one person fitting that description, we know the subject has that attribute. Otherwise he doesn't. This is the method used by Hoffman and Miller [23] and presented in the previous chapter.

It was quickly perceived that placing a minimum query set size restriction may make compromise more difficult but cannot prevent it. Karpinski [25] realized that if you "or" in some mutually exclusive set of criteria which is known to have a large count the size restriction can be bypassed. This idea of "padding" the query set is the basis of Schlörer's tracker technique. It is not necessary to define a subgroup which consists of only one person to compromise the database. If we can define two groups which differ by only one person, this works as well. What was not immediately clear was how easy it would be to devise a formula which would bring about compromise in this manner.

Schlörer [30] first introduced the tracker in 1975 for use with systems allowing characteristic-specified

queries using the boolean operators AND ($\cdot$), OR (+), NOT (-). The system is assumed to have minimum and maximum query set size restrictions. The maximum size control is necessary if the operator NOT is allowed to prevent compromise by taking the complement of the characteristic formula. One method proposed by Schlörer is this: suppose we have a certain amount of preknowledge about an individual's attributes. Let us denote this knowledge by $A = a_1 \ldots a_k$. Also suppose this information is enough to uniquely define the individual, i.e., $\mathrm{COUNT}(a_1 \cdot a_2 \cdot \ldots \cdot a_k) = 1$. We now separate our preknowledge into two sets $A_1 = \{a_1 \ldots a_i\}$ and $A_2 = \{a_{i+1} \ldots a_k\}$ such that $\mathrm{COUNT}(a_1 \cdot a_2 \ldots a_i)$ lies within the restricted range of answerable queries. We can now construct a tracker $T = (-(a_{i+1} \cdot a_{i+2} \cdot \ldots \cdot a_k)) \cdot (a_1 \cdot a_2 \ldots a_i) = (-A_2) \cdot A_1$. The tracker is also answerable and the set of records defined by it is all the records in the query set of A_1 except that of the individual we are tracking. This record can then be compromised by asking the query $\mathrm{COUNT}((T+x) \cdot A_1)$, where x is an unknown property of the record. If the answer to this query is the same as $\mathrm{COUNT}(A_1)$ then the individual has the property x. If the individual lacks the property the answer is $\mathrm{COUNT}(A_1) - 1 = \mathrm{COUNT}(T)$. The reason for this is that if the individual has property x he will be included (with others) in the query set of $(T + x)$ and the intersection of this set with that of A_1 will be precisely the query set of A_1. Figure 2.1a

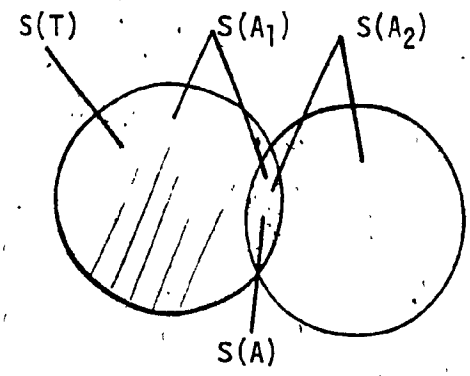
uses Venn diagrams to illustrate this point.

Another tracker building technique proposed by Schlörer [30] uses a mask M such that $COUNT(M \cdot (-A_2) \cdot A_1) = 0$. A tracker can then be constructed having the formula $(-(A_2 + M)) \cdot A_1$. The mask adds a disjoint set of records to the query set of $(-A_2) \cdot A_1$ thus "padding" the set to allow the query to be answered. The Venn diagram for this method is given in Figure 2.1b.

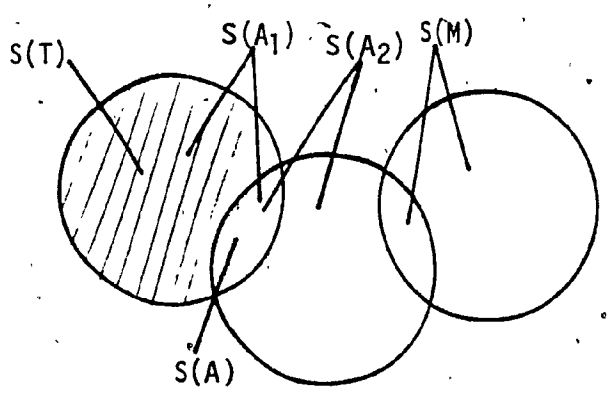
The trackers presented above require some preknowledge about an individual. In fact, they require enough preknowledge to isolate an individual's record. We can call these "individual trackers". A different individual tracker is required to isolate each record. Denning, Denning and Schwartz [12] extended Schlörer's work and devised a so-called "general tracker". Let us call the minimum query set size restriction Z_{min} and the maximum $n - Z_{min}$ where n is the number of records in the database. A general tracker is any characteristic formula T such that $2 \cdot Z_{min} \leq COUNT(T) \leq n - 2 \cdot Z_{min}$. The database can be compromised in the following manner: suppose we have asked a query $q(C)$ where C is some characteristic formula and q represents any query type (COUNT,AVG,SUM, etc.). If $COUNT(C) < Z_{min}$ this query will not be answered. We can determine the answer using the equations:

$$Q = q(T) + q(-T) \tag{2.1}$$

$$q(C) = q(C + T) + q(C + (-T)) - Q . \tag{2.2}$$



a) Method 1



b) Method 2 using mask M

S(C) denotes the set of records satisfying the characteristic formula C.

Figure 2.1 Tracker Compromise

All the queries on the right-hand side of these equations are answerable since the tracker and its complement are by definition answerable and $COUNT(C) < Z_{min}$ so $COUNT(C + T)$ or $COUNT(C + (-T))$ will be less than $n - 2 \cdot Z_{min} + Z_{min} = n - Z_{min}$. Similarly, if $COUNT(C) > n - Z_{min}$ the following equation will determine $q(C)$:

$$q(C) = 2Q - q((-C) + T) - q((-C) + (-T)) . \qquad (2.3)$$

The general tracker works on the principle that the record defined by C will be included in the query set either of T or $-T$ since they are disjoint and cover the entire database. Let us suppose, for instance, that the query set of T, denoted $S(T)$ includes the record we are tracking, defined by $S(C)$. Then $S(T + C) = S(T)$ but $S((-T) + C) = S(-T) + S(C)$. The effect of equation 2.2 is to add the answer for $S(T)$ and $S((-T) + C)$ and subtract $S(T)$ and $S(-T)$ leaving the answer for $S(C)$. For example, consider this instance of equation 2.2 and assume $COUNT(C) = 1$:

$$COUNT(C) = COUNT(T + C) + COUNT((-T) + C) - COUNT(T) - COUNT(-T) .$$

The equation then reduces to:

$$COUNT(C) = COUNT(T) + COUNT(-T) + 1 - COUNT(T) - COUNT(-T) = 1 .$$

Clearly the general tracker is a powerful inference technique. Its greatest advantage is that the same tracker can be used to compromise any record in the database.

Any formula with a query set size in the restricted subrange $[2Z_{min}, n - 2Z_{min}]$ will suffice. In most databases this can be fulfilled by the formula sex = male.

One further result recently given by Denning and Schlörer [13] concerns the amount of work needed to find a tracker. They give an algorithm for finding a tracker in $O(\log_2 M)$ queries where M is the number of distinct records possible. The only knowledge required of the user is what the different values of each domain are. This study shows that trackers are indeed a considerable threat to statistical database security and query set size restrictions are not effective in preventing compromise.

2.3 Combinatorial studies

With the statistical inference problem having been formulated and query set size controls investigated, a number of researchers started to study the problem using systems of linear equations to solve for unknown values. Most of these studies attempted to quantitatively define conditions which would assure the security of a database. Inferring individual values from a series of queries is a problem which lends itself quite naturally to the application of linear systems. Most of the studies use key-based queries and simply change the keys in successive queries and solve for the various unknown values.

The first paper to address the problem in this manner was written by Dobkin, Jones and Lipton [16]. They studied key-based queries which consisted of exactly m keys

and added the further restriction that no two queries may overlap in more than r positions. Under such a system, they investigated the behavior of the quantity $S(n,m,r,\ell)$, the smallest number of queries needed to compromise the database. There n is the total number of records in the database and ℓ refers to the number of records whose values are already known to the user. The following example is given by Dobkin, Jones and Lipton to illustrate this concept [16,p.101]:

Example. $S(n,3,2,0) \leq 4$, $n \geq 4$.

Let the four queries be:

$$Q_1 = X_1 + X_2 + X_3$$

$$Q_2 = X_1 + X_2 + X_4$$

$$Q_3 = X_1 + X_3 + X_4$$

$$Q_4 = X_2 + X_3 + X_4 .$$

Then X_4 can be found as $\frac{1}{3}(-2Q_1 + Q_2 + Q_3 + Q_4)$. The study then derives certain conditions which maximize S.

Similar studies using this model have also quantified the amount of security obtained under various conditions [7,29]. Alagar and Blanchard [1,4] identified a "forbidden query set" consisting of the smallest subset of queries without which compromise is impossible. Consequently only queries belonging to this set need to have their responses perturbed to protect the database. Reiss studied

median queries involving exactly m elements and found compromise was possible with $O(\log^2 m)$ queries for some arbitrary element and in $O(m)$ for a specific element [28]. Kam and Ullman [24] used a somewhat unusual model where keys are represented by a sequence of k bits and a query is a specification of $s \leq k$ bits. They concluded that if the range of values for a database is unrestricted this type of query cannot compromise any database. This is a somewhat unrealistic assumption. If the range is restricted to some finite quantity d, compromise is possible. It is not clear if this rather abstract model is applicable in practice.

Several studies on characteristic-specified queries have also attempted to define criteria which assure the confidentiality of compromisability of a database. Haq [20,21] determined a number of complex conditions which provide a means to check if a database is secure. These conditions include information about the user's supplementary knowledge of the database, which is difficult if not impossible to determine in practice. Using a different approach, Chin [5] studied databases which refuse to answer queries involving fewer than two records. He found that if the existence of just one record is known, a user can determine the existence of all the other records. Similarly, if the data value for any record is known, values for all the other records can be deduced. This is not at all surprising since the database answers queries involving two records.

All the user has to do is include the known record as one of the two. Chin then goes on to provide a graph-theoretical model of the database whereby each record is a vertex and edges exist between vertices if and only if there is a characteristic which isolates those two records. He proves that compromise is possible if either the graph has at least one odd cycle or there exists a characteristic C such that COUNT(C) is odd and at least three. This is an interesting approach but suffers from the drawback of many of these studies in that it provides a means of checking whether or not a database is secure and not how to secure the database.

Finally it was shown by Demillo, Dobkin and Lipton [8] that even if a database system does not respond with true answers it can be compromised. This rather surprising result should not be misleading. The database does give an exact value from some record; it is not necessarily the correct answer to the query, however. The queries studied were key-based median queries which specify a set of r records. Thus all queries were of the form: "What is the median salary of {S}?" where S denotes a set of r employees. Suppose this system does not respond with the value which is actually the median of the r values, but rather selects any value from the set at random. Demillo, Dobkin and Lipton prove that even if the overlap between queries is limited to one record, it is possible to compromise the database. The method used involves asking

m queries about the first m - 1 records, each query overlapping another by one record. Two of the queries must return the same answer by the pidgeon-hole principle (if p + 1 objects are placed in p containers, some container must have two objects). It remains merely to inspect the two queries and find the one record common to both. While this type of "lying" may not secure a database actually distorting the values to a small degree can be effective, as we shall see in the next section.

The studies on combinatorial inference have provided some insight into statistical database security. The results of these studies, however, are largely negative in that they show how easily compromisable various database models are, or at least passive in that they provide a means of checking the security of a system and not ensuring it. Most of the models are not of much practical use either, especially the key-based ones. Few real statistical databases allow records to be queried by key, and forcing the user to specify a fixed number of records per query is a very artificial constraint. Nonetheless, the studies do have theoretical interest.

2.4 Perturbation

In light of the studies which showed how easily databases could be compromised even with query set size and overlap controls, distorting the responses is an attractive method for securing statistical databases. Statistics requested of the system are randomly perturbed

in such a way that the expected values are the true values but a degree of uncertainty is introduced. This leads to large errors when inference techniques such as trackers and linear systems are used in an attempt to compromise the database.

The first such method described in the literature is called random-rounding. Nargundkar and Saveland [26] proposed the technique whereby statistics are rounded to a multiple of some "rounding base". The decision to round any value up or down is determined by a random number. For example, suppose the true answer to a query is 57 and we have chosen a rounding base of 5. We then generate a random value between zero and one. If this value is greater than .5 we round the answer to 60, otherwise we round down to 55. The user, upon receiving the response 55, can only determine that the true value lies somewhere between 51 and 59. Unfortunately, true random rounding cannot effectively secure the system since the same value may be rounded up for one instance of the query and down when the query is asked again. By averaging a number of responses to the same query, it may be possible to determine that value. The technique of pseudo-random rounding solves this problem by determining one random value for a given characteristic formula or response set. Thus the answer to any given query is always the same.

Fellegi and Phillips [18] outlined this and other

considerations needed to produce safe statistics using random rounding. Their procedure was implemented on data from the 1971 census. One such consideration is that averages should be maintained. If the original data shows 3 people earning a total of $63,000 or an average of $21,000 each, we should not round these figures separately resulting, for example, in 5 people earning $60,000, an average of $12,000 each. Instead, we would like to show either 5 people earning $105,000 or 0 people earning $0. Another source of problems is rounding totals and subtotals. If a series of values are randomly rounded and these figures then summed, the total may be well outside the desired range if we happen to round the individual values more often in one direction. Let us consider an example to illustrate this point. Suppose we have the following unrounded data: 12, 4, 7, 22, 9 giving a total of 54. If we rounded these values and then totalled we might get: 10, 5, 5, 20, 5 for a total of 45. To avoid producing large errors in subtotals and totals, these should be rounded separately, giving either 50 or 55 for our example. Percentages must be given according to the rounded values instead of the original ones. Otherwise they may reveal some information about the true values. Finally, Fellegi and Phillips note that the same queries should give the same answers, as we mentioned earlier. They proceed to present measures which are designed to achieve these objectives. This study shows that, if carefully done,

random rounding can be an effective security method.

Another similar technique releases ranges in response to COUNT queries. The user cannot be sure exactly where in the range the true response lies in such a system. Haq has shown that by giving approximate answers to queries compromise is harder but still possible [22]. This is not an irrelevant result since the effectiveness of a security method is directly related to the amount of effort needed to compromise the database. In this case the intruder must try to narrow the ranges by asking a cleverly devised sequence of queries. Alagar et. al. and Blanchard [1,4] have implemented a scheme whereby range responses are given to COUNT queries and true responses are given to SUM queries. He gives a formula for the probability that any range can be reduced, and shows that the chance of reducing a range to a single point is extremely rare and requires a large amount of work. Range responses can be subverted if dummy records are allowed to be added to the database by simply adding records with the required characteristics until the system responds with the next higher range. If the number of male lawyers, say, is given as "50-60" and, after adding 4 dummy male lawyer records, the response becomes "60-70" we know there were 56 originally. Most databases do not allow this and range responses can then provide a high level of security without producing misleading statistics.

The final perturbation method which has been studied

is that of associating some random weight with data values and thus distorting them. For instance, one could choose a random number between .7 and 1.5 for each value in the query set and multiply each value by its appropriate weight. If the query asks for the sum of the values, the system produces a "weighted sum". Schwartz, Denning and Denning used this method [32], but they chose to associate the weights with the position of a key and not the key itself. Therefore by asking a key-based query with the same keys in different positions, it is possible to determine the weights and thus the data values if any of the values are known beforehand. Recently Beck [3] has implemented a much more sophisticated model for distorting statistical data. He uses perturbation factors which are random but not necessarily uniform. The variances of the weights are chosen to conform with the known statistics of the data values, and can be altered to provide different levels of security. The effect of this is to prevent the user from estimating a true value by asking a series of queries and determining the variation (perturbation) which has been applied to it. Beck's method gives accurate results and can without much difficulty force a user to pose 10^9 queries to infer information. There are no restrictions placed on the queries regarding size, overlap, or type.

Data perturbation has been shown to be a reasonably effective means of protecting a statistical database. The various methods presented in this section must be implemented

with great care, however, to prevent users from determining values within an acceptable error tolerance.

2.5 Threat monitoring

One means of preventing unauthorized access of individual information is to monitor the queries asked of the system in order to determine if any sequence of queries would lead to statistical inference.

Hoffman and Miller [23] first suggested the idea of keeping a log of queries to check for "bursts of activity" or queries identifying small sets of records. Fellegi [17] showed that it is theoretically possible to correlate a new query with all the previously released information and determine whether disclosure of individual information could occur. This is a rather monumental task which requires taking all possible unions and intersections of the sets of information given out by the system.

Schlörer [21] suggested a scheme which sets up standard classifications for each variable. Each classification x_i consists of one or more values in the domain of that variable, and is associated with the relative frequency of the records occurring under that classification, p_i. One can then determine the identifying power of any characteristic formula by taking the product of the p_i's. If this falls below some predetermined value the query is cancelled. Schlörer also suggests monitoring only the "dangerous" variables, meaning ones which may be easily determined from outside sources and used to isolate records. Dangerous variables

include sex, age, marital status, etc., whereas innocent variables are those not likely to be used as preknowledge about a subject such as laboratory findings or political contributions.

Threat monitoring suffers from several drawbacks. Primarily, it requires a lot of overhead. Maintaining a log of all inquiries made of the system necessitates using a large amount of storage, which must be made available to the monitoring system. The log cannot be infinite and so will only monitor queries made over a certain period of time. A clever user who knows of the existence of a threat monitor may be able to ask queries over a longer period than that for which the system can keep track. A system will certainly have to monitor queries for more than one job at a time. Another reason for this is that two or more users may conduct seemingly innocuous inquiries and later correlate their results. It is nearly impossible to detect this type of threat. Another problem is that threat monitors may restrict the release of useful information. If too many queries are denied because they may potentially be used to infer individual information, the system may no longer be serving its primary function - to release statistics about its population.

If a threat-monitoring device is to be used in a statistical database, it cannot be too elaborate or it runs the risk of hindering the flow of useful information. Schlörer [31] has suggested that such a system be used in conjunction with random-rounding. Each provides some

deterrent against inference and will increase the amount of work and time needed to compromise a database.

2.6 Response set controls

The final type of inference control mechanism which has been investigated is concerned with the set of records relevant to a query. Two major methods have been proposed which alter the query set. The first uses a random sample of the records in the query set to obtain the desired statistics, while the other partitions the database into small groups which cannot be broken down further. Both methods control statistical inference by creating uncertainty about the records used to generate the statistics queried. Random sampling eliminates some records from the query set while partitioning generally adds records from partitions where some but not all of the members belong to the query set.

2.6.1 Random sampling

Taking a random sample from a database for statistical purposes was first mentioned by Hansen who reported that the U.S. Census Bureau released detailed information from a sample of 1 in 1000 forms received in the 1960 census [19]. Names, addresses and any geographic identification was deleted except for broad city-size classes within nine geographic divisions. Using information from such a sample, it would be very difficult to accurately identify an individual. Even if one of the records matched all the preknowledge a user had, there is only a 1/1000 chance that the person being looked for was actually included in the sample. The

probability of misidentification is great and a lot of work is needed to identify at best a small percentage of records. The system has apparently worked well, but unfortunately is only applicable to very large databases.

Denning has implemented a random sampling procedure for small databases [15]. The sample is generated for each query when determining the query set. A sampling probability p is fixed beforehand, indicating the proportion of records included in the sample. Each record has an identifier consisting of several bits. When a record i is found to be in the query set, a selection function $f(C,i)$ determines whether the record is to be kept in the sample. This function matches the identifier for record i with another function $g(C)$ which maps the characteristic formula into a sequence of bits. If the two bit patterns match the record is excluded from the sample. The function $g(C)$ is designed to match the record identifiers at random with a probability of $1 - p$.

Denning shows that statistics generated using this method are fairly accurate for $p = .9375$ and databases of size 100 and 1000. Smaller p yields less accurate results, as would be expected. The major question is one of security. Random sample queries introduce an uncertainty about the composition of the response set, but can this uncertainty be reduced or eliminated?

Random tracker attacks were made on the random sample queries, but the results reported are virtually meaningless.

Mean relative errors are given for values inferred with the trackers, and these are quite large, from 200 to over 700 percent. Some trackers, however, may be off by a tremendous amount while others may accurately infer individual information. Denning gives us no idea how many trackers inferred values to within, say 10%.

One problem addressed by Denning is that of small query set sizes. If the query set is small there is a chance that all the records will be included in the sample. This chance, of course, increases with p. Since a large value of p is needed to produce accurate statistics, small query sets are particularly vulnerable. To combat this, Denning suggests lowering the value of p for these queries, which seems like an awkward technique and will produce highly innacurate results, or to make a minimum query set size restriction.

Another major problem with this particular implementation is that the sample selected is based on the characteristic formula C. Thus if an equivalent formula is queried it will generate a different random sample of the same records. Denning suggests that some normal form for C could be used to form the sample, but that the problem of reducing logical formulas to normal form is intractable. Error removal is then possible by averaging responses to queries which specify the same query set or by averaging estimates from queries about disjoint subsets of a query set. Denning shows that it would take a large number of

queries to accurately determine individual values in this manner. While this would discourage a manual attack, it certainly could be performed with a computer. She suggests some form of threat monitoring to detect such error removal attempts. It seems to us that a simpler method for preventing this type of attack is to make the function g dependent upon the query set size and not the characteristic formula. Then any equivalent formulas would produce the same sample since they involve the same number of records. It is not clear if this would prevent disclosure by asking queries about disjoint subjects, however.

Empirically, random sampling appears to provide effective inference controls. It can produce accurate statistics and the samples are generated at response time providing up-to-date information. It is not clear, though, whether security can be guaranteed under such a system.

2.6.2 Database partitioning

The idea of separating a database into disjoint groups was introduced by Conway and Strip[6]. They suggested using classes instead of values for the various attributes. Each class would contain several records and it would not be possible to isolate any record within a class. For example, let us consider the attribute "salary". One class for this attribute might be class 15,000, including all salaries in the range 15,000 to 20,000. The condition "salary = 16,000" would be interpreted as true if the class of "salary" is the same as the class to which 16,000

belongs (namely class 15,000). This could be implemented interpretively, involving a table look-up of the class definitions for each access, or the class to which a value belongs could be determined once and stored directly in the record. Conway and Strip do not give a detailed explanation of the method, but are quite enthusiastic about its potential for securing statistical databases. They state that "there does not appear to be any way, even with repeated queries that unauthorized information could be extracted from a field protected in this manner."[4]

A more detailed partitioning proposal was made by Yu and Chin [33]. Whereas Conway and Strip partitioned each attribute into disjoint classes, Yu and Chin propose partitioning the entire database into mutually exclusive groups. The idea is the same: If a query refers to any member of a group, it is answered for all members of the group. This way no record may be isolated, no matter what queries are asked. The critical problem is to create useful partitions, i.e., ones which contain records which are as similar in nature as possible.

Yu and Chin give an algorithm for producing the partitions. The algorithm and its drawbacks are discussed in detail in the next chapter. They also give methods for including new data as well as updating and deleting old

---

[4]Conway & Strip, Selective Partial Access to a Database (ACM 76, Houston), p.89.

data in the system. These methods involve mainly flagging any changes but not using the new information until enough updated data has accumulated to warrant redefining a partition.

This thesis presents the implementation of two partitioning algorithms for securing a statistical database. The partitioning technique is the soundest one in theory which exists to protect a database against statistical disclosure. We will show that useful statistics can be produced using this method and the partitioning can be achieved at a low cost both in terms of time and overhead.

CHAPTER THREE

RECTANGULAR PARTITIONS

3.1 Introduction

The first method we have used to partition a database is derived from the 1977 paper of Yu and Chin [33]. In that paper they describe how a database can be partitioned through an initial mapping onto a matrix of all possible records. This matrix can subsequently be partitioned into disjoint rectangular regions, each of which contains at least some minimum number of records. In this chapter we will first describe Yu and Chin's method and its drawbacks. Then we will present a superior algorithm that is almost optimal for partitioning a database. Complexity analysis is given and selected test results are presented.

3.2 Yu and Chin's partitioning algorithm

Let us assume that we have a database consisting of N records, each of which has values in K domains or attributes. Call d_i the size of the ith domain; that is, there are d_i different values possible for the ith domain. For example, if the first attribute is SEX, then $d_1 = 2$.

It is possible to construct a K-dimensional matrix of size $d_1 \times d_2 \times \ldots d_K$, each cell of which represents one possible set of values that a record in the database can have. Furthermore, any valid record can be mapped onto a cell in this matrix. Consider a record r_j; with

values $j_1, j_2, \ldots j_K$. This record is associated with the cell having coordinates $[j_1, j_2, \ldots j_K]$ in the matrix. When all the N records have been placed in the matrix, each cell will have a count representing the number of records at that position.

We must now partition this database matrix into disjoint regions so that each region contains at least some minimum number of records given by our threshold, $t \geq 2$. Some cells may already contain t records, and these are valid partitions by themselves. Some cells may have no records associated with them, while others will contain some records but less than the threshold. The cells in the latter group must be merged, somehow, with surrounding cells to create the desired partitions.

The procedure suggested by Yu and Chin is to merge a nonempty cell (with less than t records) with one of its neighbouring cells by combining all the cells with those two adjacent values in that domain. This process is repeated until the cell has at least t records, and is performed for each nonempty cell with less than t records.

Consider a two-dimensional example; that is, there are just two attributes per record. Let $y_1, \ldots y_n$ and $z_1, \ldots z_m$ denote the possible values in each domain. Thus, we start with a two dimensional matrix B containing nm cells. Once the matrix is filled with all the records, any cell $B(i,j)$ contains all records (u,v) such that $u = y_i$ and $v = z_j$. If this region is nonempty and has fewer than

t records, Yu and Chin's algorithm merges this region with its neighbours in the arbitrary order $B(i,j+1)$, $B(i+1,j)$, $B(i,j-1)$ and $B(i-1,j)$ until the resulting region contains at least t records. Suppose $B(i,j)$ is merged with $B(i,j+1)$ and $B(i+1,j)$. The new domains then become $D_1 = \{y_1, \ldots y_j \cup y_{j+1}, \ldots y_n\}$ and $D_2 = \{z_1, \ldots z_i \cup z_{i+1}, \ldots z_m\}$. See Figure 3.1. This procedure combines entire rows or columns instead of simply combining the cells needed to make a valid partition. A cell which in itself is a valid partition (i.e., containing $\geq t$ records) may be forced with its adjacent cells to form a larger sized partition with number of records much greater than t. This might also include several empty cells in it thus introducing large errors in the computed statistics. Furthermore, when carried to its completion, this merging process can cause the virtual collapse of the database into a far fewer number of partitions than the theoretical maximum of $\lfloor N/t \rfloor$. See Figure 3.2.

Yu and Chin state that the complexity of the algorithm is proportional to the number of initial regions in the matrix, since each region is examined once. We believe that, although the algorithm is fast and relatively simple, it is not very useful in practice as a partitioning scheme. It is more desirable to have an algorithm which primarily maximizes the number of partitions and secondarily minimizes the area covered by those partitions (that is, includes as few empty cells as possible), even if it is more complex. It should be noted that Yu and Chin did not attempt to implement their

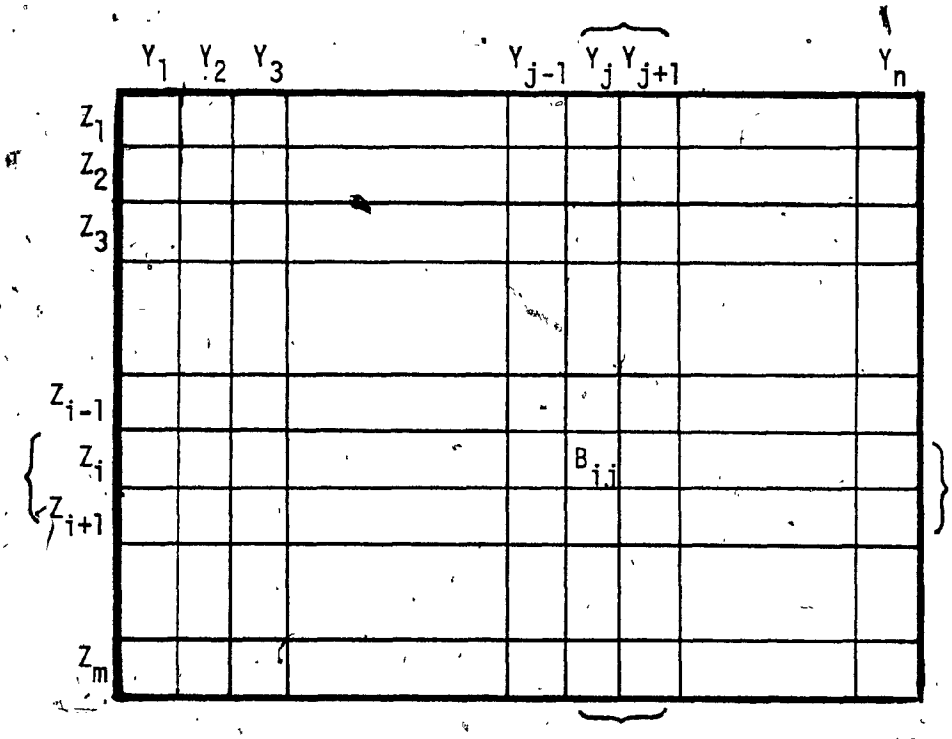


Figure 3.1 Merging B_{ij} with its neighbours $B_{i,j+1}$ and $B_{i+1,j}$.

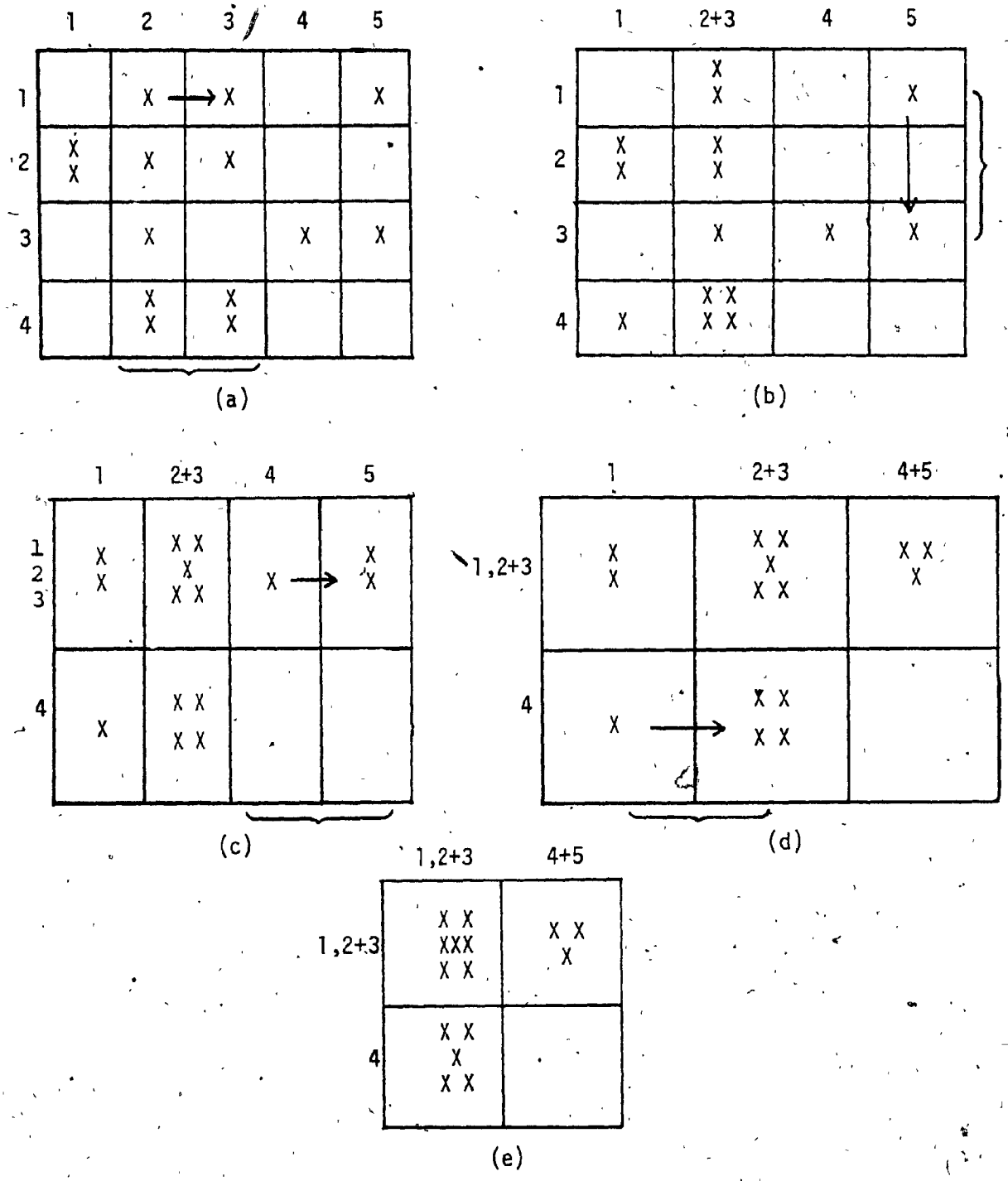


Figure 3.2 Merging a sample database by Yu and Chin's method. Arrows indicate which cells are being merged. Brackets indicate which values are being merged. Each record is represented by an X. For this example: k=2, t=2, N=15, d_1=4 and d_2=5.

algorithm nor test its performance. The article [33] was, however, the first and as far as we know the only paper which even proposes a database partitioning algorithm for the purpose of safeguarding against statistical inference.

3.3 Rectangular partitioning

We have designed an algorithm which partitions a database into groups such that each group contains at least t records. If the database is mapped onto a k-dimensional matrix, our algorithm obtains a disjoint set of rectangular regions (rectangular parallelopipeds for $k > 2$) which cover all the non-empty cells of the matrix. The regions are formed subject to the following constraints:

1) each region has at least t records;
2) the number of partitions obtained is maximum, and
3) the sum of the areas (volumes) of the partitions is minimum (i.e., the number of empty cells included in the rectangles is minimum).

The algorithm, consisting of two phases, partitions a two-dimensional matrix into disjoint rectangular regions subject to the above restrictions. It can then be applied repeatedly for databases of higher dimensions by taking the rectangles formed in one application of the algorithm as one axis of a new two-dimensional matrix, and the values of the next domain as the other axis. This process will be described in detail later in this chapter.

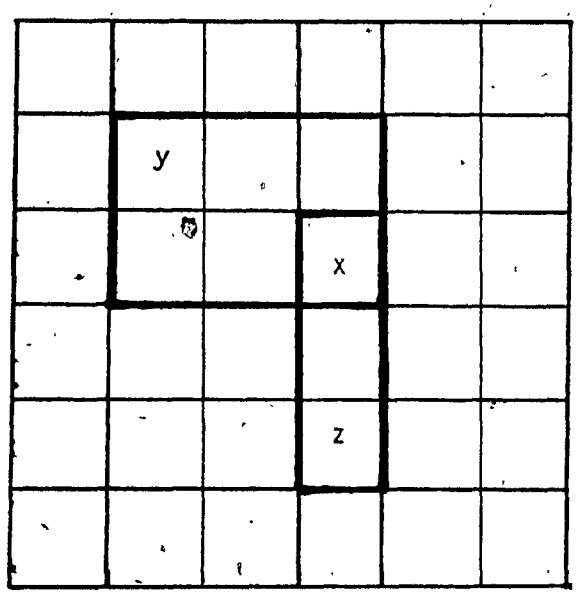
We have used a graph-theoretical approach to solve the rectangular partitioning problem. Thus, we consider

each nonempty cell in the matrix as a vertex of the graph. The first phase of the algorithm then defines the graph by constructing adjacency lists for all the vertices. Two vertices p and q are defined to be adjacent if and only if a rectangle can be formed which encloses the two cells corresponding to p and q with no other nonempty cell included in this rectangle. See Figure 3.3. The size (or area) of a rectangle so formed will be the weight of the edge (p,q). The set of all vertices found to be adjacent to p can be called the "nearest neighbours" of p. It is these vertices which are candidates to be included in the rectangle of which p is a member.

Once the adjacency lists have all been constructed, the second phase of the algorithm forms the rectangles using a region growing process. It first selects the vertex which has the most records associated with it or two adjacent vertices such that their edge has minimum weight. The algorithm then proceeds to grow a rectangular region of minimum size until the total number of records enclosed by that region exceeds or equals the threshold, t. By selecting a vertex not included in any of the previously formed rectangles and repeating the process, we form more rectangles until no more vertex remains to be covered.

3.4 Optimality considerations

Obtaining an exact solution meeting the requirements specified above, i.e., that each partition contains at least t records, the number of partitions be maximum and



x and y are adjacent

x and z are adjacent

y and z are not adjacent

Figure 3.3 Adjacent Vertices.

their total area be minimum, is hard in the sense that any algorithm attempting to find such an optimal covering will invariably be required to do an exponential amount of work proportional to the input size. We have written a backtracking algorithm which illustrates this point. The algorithm assumes that phase I has been completed, hence the adjacency lists are formed for each vertex. The vertices all have a tag field, whereby they can be marked as used in a partition or not. The function wt(x) returns the number of records associated with a vertex x or the number of records included in a rectangle x. The variables "bestsofar" and "bestarea" store the number of rectangles and the area covered by the best partitioning found by the algorithm at any given time. The level of backtracking is given by ℓ and the set of vertices to be examined at that level is S_ℓ.

<u>Algorithm OPTPART</u>

1. Unmark all vertices. $\ell \leftarrow 0$.
2. For each vertex v such that $wt(v) \geq t$, mark (v) and add a rectangle consisting of v to the set of rectangles. $S_\ell \leftarrow \phi$. $\ell \leftarrow \ell+1$.
3. $S_\ell \leftarrow$ {all v such that v is unmarked}.
4. If $\Sigma wt(v)$ for all v in $S_\ell < t$ then go to step 11.
5. Choose an element from S_ℓ, call this v. Delete v from S_ℓ. Mark (v). Total weight $\leftarrow$ wt(v). $\ell \leftarrow \ell+1$;
6. $S_\ell \leftarrow$ {all w such that w is in the adjacency list of

v and w is unmarked}.

7. If $S_\ell = \phi$ execute step 11.

8. Choose an element from S_ℓ, call this w. Delete w from S_ℓ. Mark (w).

9. If total weight + wt(w) < t then include w in the current rectangle, set $\ell \leftarrow \ell+1$, $v \leftarrow w$ and return to step 6.

10. (Total weight + wt(w) $\geq$ t).
Add current rectangle including w to the set of rectangles, set $\ell \leftarrow \ell+1$ and return to step 3.

11. For each unmarked vertex, include it in the nearest rectangle.

12. If the number of rectangles > bestsofar or number of rectangles = bestsofar and area covered < bestarea then save this set of rectangles and purge any previously saved set.

13. Unmark (v). $\ell \leftarrow \ell-1$. If $\ell \geq 0$ then return to step 7.

Since algorithm OPTPART considers each vertex in the set S for all levels ℓ, it will eventually form all the possible sets of rectangles and simply choose the best one in step 12.

In analyzing the cost of this algorithm we realize that the height of the backtracking tree is m-1, where m is the number of vertices found in phase I, since each vertex must be examined to determine a set of rectangles which defines a partitioning of the database. Furthermore, at

each level i of the tree there have been i vertices already looked at and marked, leaving $m-i$ vertices which may be included in the next set S_i. Thus a node at level i will have at most $m-i$ sons. The maximum size of the tree, and consequently the cost of the algorithm then becomes $m \cdot (m-1) \cdot (m-2) \cdot \ldots \cdot 1$ or $m!$ The algorithm is therefore exponential with time complexity of $0(m!)$.

In an attempt to reduce the cost and yet obtain an almost optimal solution to our problem, we have designed an algorithm which produces a "nearly optimal" partitioning in time $0(x^2 \log_2 x)$ for most input and in time $0(x^3)$ for some rare kind of input database in which the distribution of values is concentrated, where x is the number of groups formed.

3.5 Constructing the adjacency lists

The first phase of the matrix partitioning procedure transforms the matrix into a graph. This involves scanning the neighbourhood around each nonempty cell to find all the vertices adjacent to it. First we will informally describe this process, then we give a formal description of the algorithm. In the actual implementation of the algorithm, the adjacency lists are kept as a linked list wherein each node corresponds to a vertex and contains the identifying coordinates of the vertex (i and j), the weight of the vertex (the number of records associated with the cell (i,j)), a tag field to mark the vertex used or unused, a pointer to its adjacency list, and a link to the next node.

The nodes in the adjacency list simply contain a pointer to the vertex they represent, and a link field to the next list node.

3.5.1 General description of the algorithm for Phase I

We are given a two-dimensional matrix of size $D_1 \times D_2$, where D_1 and D_2 are the number of different values in two domains of the database. The value in each cell is the count for that cell, some nonnegative integer. For example, suppose the value of cell (i,j) is 2. It means there are two records in the database with values i for the first domain and j for the second.

Let us consider any nonempty cell (i,j) for which we want to construct its adjacency list.

We must find all rectangles which include (i,j) and one other cell only. The first step in this procedure is to scan the i-th row and j-th column in each direction, until a nonempty cell or the edge of the matrix is found. Suppose we find four cells in this manner. Call them $(i+a_1,j)$, $(i-a_2,j)$, $(i,j+b_1)$ and $(i,j-b_2)$. These cells define the maximum area to be searched, namely all cells (x,y) where $i-a_2 < x < i+a_1$ and $j-b_2 < y < j+b_1$. Any rectangle which includes (i,j) and some nonempty cell outside this region must also include one of these four cells and, hence, cannot be adjacent to (i,j). Figure 3.4 illustrates this point. The area to be searched can be broken into four quadrants. Each of these quadrants can then be searched by rows starting with the row nearest to i and working away from (i,j).

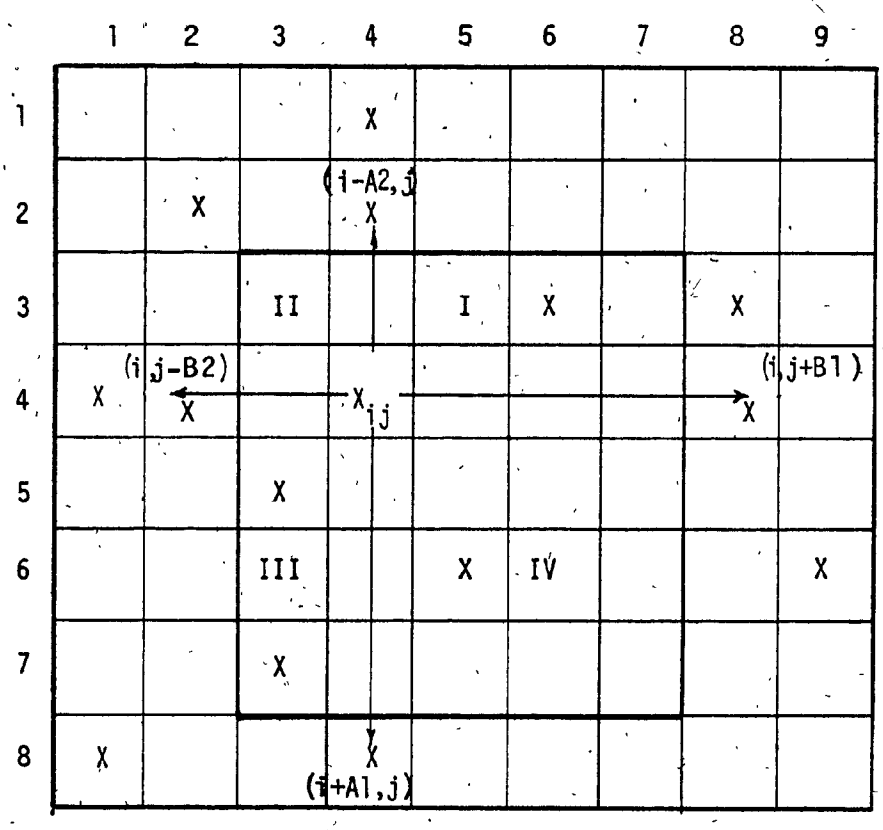


Figure 3.4 Finding the four quadrants to be searched for the adjacency list of cell (4,4).

As soon as a nonempty cell is encountered it is added to the adjacency list of (i,j), and it further limits the area to be searched in that particular quadrant. The newly found cell effectively "blocks" all cells beyond it from (i,j). Figure 3.5 shows the areas blocked by a cell in each quadrant. Once all the quadrants have been searched in this manner, the adjacency list for (i,j) is complete and all the neighbours of (i,j) have been found. This process is repeated for all m vertices. Although we have tried to limit the areas to be searched for each vertex, it can still be rather costly, so we have introduced two modifications to reduce the search time and to avoid searching the same area too many times.

The first modification comes from the fact that, if a vertex (x,y) is found to be adjacent to (i,j), then (i,j) should also be included in the adjacency list of (x,y). Thus, every time we find a vertex which is adjacent to another, we insert them both into each others' lists. This eliminates the need to search two of the quadrants for each vertex. The search starts at row 1, column 1 of the matrix and proceeds across the row until column D_2, then starts at the next row and proceeds as before, i.e., in a top down left to right fashion. When scanning the area around a vertex (i,j) we need never consider the two quadrants above (i,j) for adjacent vertices. Any cell (x,y) in those quadrants which is adjacent to (i,j) will have found (i,j) during its own search and (x,y) will

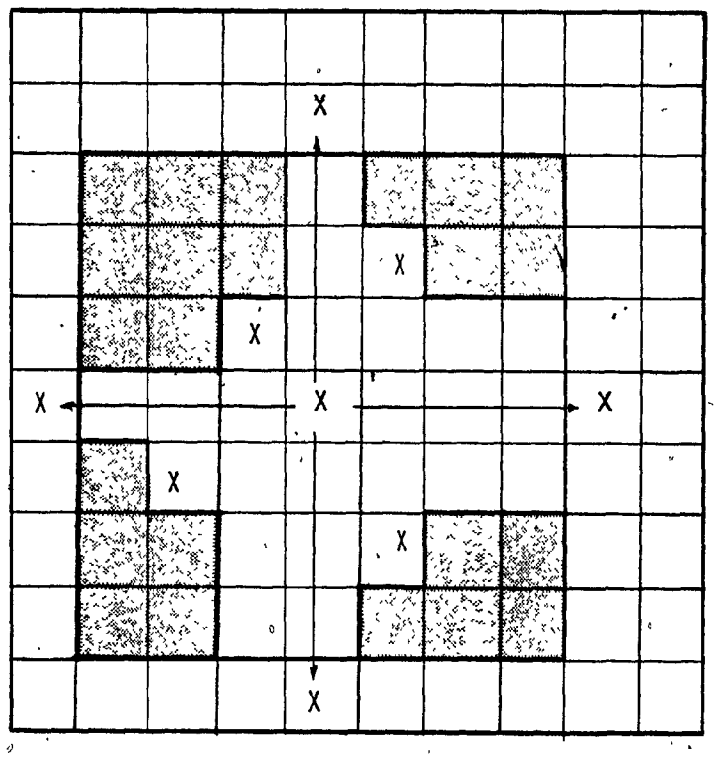


Figure 3.5 Areas blocked by a cell in each quadrant.

already be in the adjacency list of (i,j).

Although this modification will reduce the search time considerably, it is still a possibility that each search will take $O(D_1D_2)$ time. This can happen if, for instance, all the nonempty cells occur in the top right portion of the matrix. Thus the worst case time complexity is still $O(mD_1D_2)$ where m is the number of nonempty cells. In order to further reduce the complexity we introduce the second modification described below.

The matrix can be preprocessed in the following manner: the value of each empty cell can be changed to indicate where the nearest nonempty cell is, in either direction along its row. Thus it is no longer necessary to search the rows from the cell in consideration downwards, but merely to inspect the cells of the column j by increasing rows until a nonempty cell is encountered. All that remains to be done is to test if the two nonempty cells pointed to by each empty cell in column j lie within the region of possible adjacent cells. Determining this region is also an easy process, requiring only that we inspect the two adjacent cells in row i, namely $(i,j-1)$ and $(i,j+1)$. These cells will either be nonempty, in which case the area to be searched is bounded by that column, or they will indicate the nearest nonempty cell in row i which then defines the maximum area to be searched. Each vertex found by inspecting column j which is determined to be within this region subsequently narrows the region as described

	1	2	3	4	5	6	7	8	9	10
1		X					X		X	
2		X			X	X		X		
3	X			X			X			
4			X		X					
5			X	-101	X	-103			X	
6	X				-403			X		
7				X	-102		X			
8				X	-104				X	
9					X					
10		X				X				x

Cells searched in finding the adjacency list for (5,5):

1. (5,6) right:=9 (5,9) added to list
2. (5,4) left:=3 (5,3) added to list
3. (6,5) right:=8 (6,8) added to list
   (6,1) not added to list
4. (7,5) left:=4 right:=7
5. (8,5) (8,4) and (8,9) not added to list
6. (9,5) (9,5) added to list

Figure 3.6 The second modification for Phase I.

earlier. We have chosen the following code to represent the nearest nonempty cells to an empty cell: the value of an empty cell will be (the distance to the nearest vertex to the left) * 100 + (the distance to the nearest vertex to the right), all negated. So, if there is a vertex at (3,2) and one at (3,10) with empty cells in between, the entry for cell (3,5) will be -305, indicating a nonempty cell three spaces to the left and another nonempty cell five spaces to the right. Figure 3.6 presents an example of this method for finding an adjacency list. The matrix is extended to include columns 0 and D_2+1 which are used if there are no nonempty cells to the left or right of an empty cell, respectively. This modification guarantees that the worst case time for Phase I will be $0(D_1D_2)$. A detailed analysis will be presented later in this section.

3.5.2 Formal description of the Phase I algorithm

Below is the formal algorithm for Phase I, which determines the adjacency lists for all vertices. Assume that initially the matrix consists of counts for each cell (≥ 0). Steps 1 to 8 perform the preprocessing of the empty cells in the matrix.

<u>Algorithm FINDADJLISTS</u>

1. rownumber ← 1
2. col1 ← col2 ← 1
3. if matrix[rownumber,col2] nonempty then add it to list of vertices
4. advance col2 to next nonempty cell or edge of matrix

5. for each empty cell from col1 to col2, calculate distance factors and assign to cell
6. if col2 $\leq D_2$ then col1 ← col2 and return to step 3
7. rownumber ← rownumber + 1
8. if rownumber $\leq D_1$ then return to step 2
9. choose next vertex (i,j) in list of vertices; if no more vertices stop
10. if matrix[i,j+1]>0 then right ← j+1
    otherwise right ← j+1+(matrix[i,j+1] mod 100)
11. if right $\leq D_2$ add (i,right) to adjacency list of (i,j) and vice versa
12. if matrix[i,j-1]>0 then left ← j-1
    otherwise left ← j-1+(matrix[i,j-1] div 100)
13. row ← i+1
14. if matrix[row,j]>0 then add (row,j) to adjacency list of (i,j) and vice versa, and return to step 9
15. if j+matrix[row,j] div 100 > left then
    add (row,j+matrix[row,j] div 100) to adjacency list of (i,j) and vice versa,
    and set left ← j+matrix[row,j] div 100
16. if j+matrix[row,j] mod 100 < right then
    add (row,j+matrix[row,j] mod 100) to adjacency list of (i,j) and vice versa,
    and set right ← j+matrix[row,j] mod 100
17. row ← row + 1
18. if row > D_1 then return to step 9

otherwise return to step 14

3.6 Forming rectangular partitions

We are now ready to start the actual partitioning of the matrix into disjoint rectangular regions. Our basic strategy is to choose the pair of adjacent vertices which are closest to each other to start a rectangle. We then continue to build the rectangle by choosing the next closest vertices until the rectangle contains at least t records. This process is repeated until no more valid rectangles can be formed.

A rectangle is stored as a record consisting of the following fields:

1. Its weight - the sum of the weights of all its vertices. This is the total number of records contained within the rectangle.

2. Its parameters xmax, ymax, xmin, ymin, where (xmin,ymin) and (xmax,ymax) are the coordinates of the top left and bottom right corners of the rectangle and any cell (i,j) with $xmin \leq i \leq xmax$ and $ymin \leq j \leq ymax$ is in the rectangle.

3. A list of the vertices contained in the rectangle.

There are several functions made use of in the algorithm to help find vertices to add to a rectangle which is not yet a complete partition. The first, Rectsize(R,q) returns the size of the rectangle formed by expanding the already formed rectangle R to include the vertex q. Rectdist(R,q) returns the minimum rectilinear distance from q to any of

the vertices in R. The boolean function Overlap(R,q) returns the value true if the rectangle formed by expanding R to include q intersects with any other existing rectangle (partition).

3.6.1 Informal description of the algorithm

The nearly optimal partitioning algorithm uses solely the adjacency lists created by FINDADJLISTS as input (as, indeed, does the optimal algorithm OPTPART, presented previously). It produces a set of rectangles defining the partitioning of the database.

The first step in this process is to select all vertices with weight greater than or equal to the threshold t, as these are all valid rectangles already. Next the adjacency lists of all the remaining vertices are searched and each pair of vertices is stored in another list according to the size of the rectangle enclosing them. Thus, if vertex (i,j) is in the adjacency list of vertex (x,y) and the size of their enclosing rectangle, given by $(|x-i|+1)*(|y-j|+1)$, is s, they will be inserted into the list of all pairs of size s. Any cell with weight greater than one is called a "pair" of size 1.

A new rectangle is formed by selecting the smallest unmarked pair and creating a rectangle consisting of that pair. The vertices are then marked as being included in a partition. Then, if the weight of the rectangle is less than t, vertices are added one by one such that the size of the new rectangle is minimum, until the rectangle has

a total weight of at least t . Theoretically, the function "Rectsize" should be used to select these added vertices, since that function will find the vertex which increases the size of the rectangle minimally. In practice, however, this tended to create long, narrow rectangles, which can isolate many other vertices, resulting in fewer rectangles actually being formed. Thus we found "Rectdist" to give a better overall performance than "Rectsize".

Once a rectangle is completed, the next smallest pair is chosen, and the process repeated. When no more valid rectangles can be formed, there will invariably be some vertices which have not been included in any rectangle. These points cannot form a rectangle themselves, either because fewer than t points remain or because no rectangle can be formed with them without intersecting one or more previously formed rectangles. We call these vertices "left-overs" and each one is simply included in the rectangle nearest to it.

One further note concerning this algorithm is in order. It may happen that a rectangle is started but cannot be completed, i.e., there are no more unmarked vertices available to be added to it. In this case, the rectangle is dismantled and all of its vertices become left-overs. It is possible that some of these vertices could be included in some other rectangle later on, but the situation where a rectangle must be dismantled is rare and that in which some of those vertices can be reused is rarer still. Furthermore, many rectangles may have to be started and aborted in order to discover

whether or not any vertices can indeed be included in another rectangle. Our results show that this situation occurred in less than five per cent of the trials, and in each case the total number of rectangles was reduced by one. Therefore, in the interests of time efficiency we convert all vertices in dismantled rectangles into left-overs.

3.6.2 Formal description of the Phase II algorithm

Here we present the formal algorithm for producing a nearly optimal partitioning of a two-dimensional matrix into disjoint rectangular regions, with cost $0(x^2 \log x)$ where x is the number of regions formed.

A pair of vertices, v and u, is chosen in step 2. In the case of pairs of size one, which consist of only one vertex, assume that u is a null cell with weight = 0. Let $wt(x)$ denote the number of records associated with a vertex x or the total number of records included in a rectangle x.

<u>Algorithm NEAROPTPART</u>

1. for each vertex v do
   if $wt(v) \geq t$ then mark (v) and output rectangle consisting of v
   else if $wt(v) > 1$ then insert v into size(1) list
   else for each vertex u in the adjacency list of v, insert v and u into list of appropriate size
2. Select the first pair of vertices v and u on the nonempty list of least size and delete from list
   if all lists empty go to step 9
3. if either vertex is marked, return to step 2

4. mark v and u and create a rectangle R consisting of these vertices
5. if wt(R) $\geq$ t then output R and return to step 2
6. find the unmarked vertex w such that Rectdist(R,w) is minimum and NOT overlap(R,w); if none exists go to step 8
7. mark (w) ; add w to R and return to step 5
8. call all vertices in R left-overs and return to step 2
9. for each left-over vertex, add the vertex to the rectangle which increases size least and is of minimum weight.

3.7 Cost analysis

We will make use of the following variables in our analysis of the algorithms FINDADJLISTS and NEAROPTPART:

n = number of records in the database

m = the number of vertices (nonempty cells)

D = the length of a side of the matrix (assume $D_1 = D_2 = D$)

x = the number of rectangles formed

y = the number of left-over vertices

z' = the number of rectangles aborted (in step 8)

z = the total number of vertices in the aborted rectangles

We can show that the following relationships between these variables hold:

r1. $m \leq N$

This is clear from the fact that each vertex has a weight of at least one. Therefore there can be at most n vertices.

r2. $x \leq N \text{ div } t$

Since each rectangle must contain at least t records a maximum of the greatest integer less than N/t rectangles can be formed.

r3. $y \leq m-x$

The least number of vertices in a rectangle is one. If all the rectangles contain only one vertex, there will be $m-x$ left-over vertices.

r4. $z \leq y$

When a rectangle is started but not enough vertices can be added to complete it, it is destroyed and all its vertices become left-overs.

r5. $z' \leq z$

This relation is true because each aborted rectangle must have contained at least one vertex.

3.7.1 Analysis of FINDADJLISTS

In order to preprocess the matrix, each row must be scanned once to find the nonempty cells. Then the empty cells are assigned the appropriate values. This requires at most two visits to each cell. Therefore the total preprocessing

cost is $2*D^2$.

Once the preprocessing has been done, the column below each vertex is scanned and one cell on either side of the vertex is examined. Thus the total number of cells examined is $2m$ plus the sum of all the cells scanned in the columns below each vertex. Each column, however, is only inspected once altogether since a column is searched below each vertex until another vertex is found. That vertex will then search the same column below it, and so on until the edge of the matrix is finally encountered. Figure 3.7 illustrates this point. The number of cells searched in this manner, then, is bounded by D^2, making the total cost of the algorithm $2m+D^2$ after preprocessing. Since m must be less than or equal to D^2 (there cannot be more than one vertex per cell) this cost is proportional to D^2, as is the preprocessing cost. This gives an overall worst case cost of $O(D^2)$ for algorithm FINDADJLISTS.

3.7.2 Analysis of NEAROPTPART

Before proceeding with the formal analysis of the second phase of the algorithm we must comment on the storage structure used for the rectangles when they are formed.

The rectangles are stored in a quaternary tree; that is, a tree with nodes of degree four or less. Since the rectangles formed are disjoint, any given rectangle must lie completely to the right, left, above or below any other. Thus a quaternary tree is a suitable and sufficient data structure for storing the rectangles. Rectangles are

	1	2	3	4	5	6	7	8	9	10
1		X			-302		X		X	
2		X			X	X		X		
3	X			X	-102		X			
4			X		X					
5			X		X				X	
6	X	X			-303			X		
7				X	-102		X			
8				X	-104				X	
9					X					
10		X			X					X

(1,5) is not searched
(2,5) is not searched by any cell in column 5
(3,5) and (4,5) are searched by (2,5)
(5,5) is searched by (4,5)
(6,5), (7,5), (8,5) and (9,5) are searched by (5,5)
(10,5) is searched by (9,5)
the edge is found by (10,5)

Figure 3.7. Searching done in column 5 by cells in that column.

inserted into the tree as they are created, according to their position with respect to the root rectangle and any others they encounter until they reach a leaf node. The branch taken at a given node depends on the relative position of the test rectangle to the rectangle at that node. This can be found with at most four calculations. Let us assume we have a rectangle R_0 in the tree and a new rectangle R_i to be inserted. The four tests required to determine the relative position of R_i to R_0 are as follows:

If $xmax(R_i) < xmin(R_0)$ then R_i is on top of R_0

If $xmin(R_i) > xmax(R_0)$ then R_i is below R_0

If $ymax(R_i) < ymin(R_0)$ then R_i is to the left of R_0

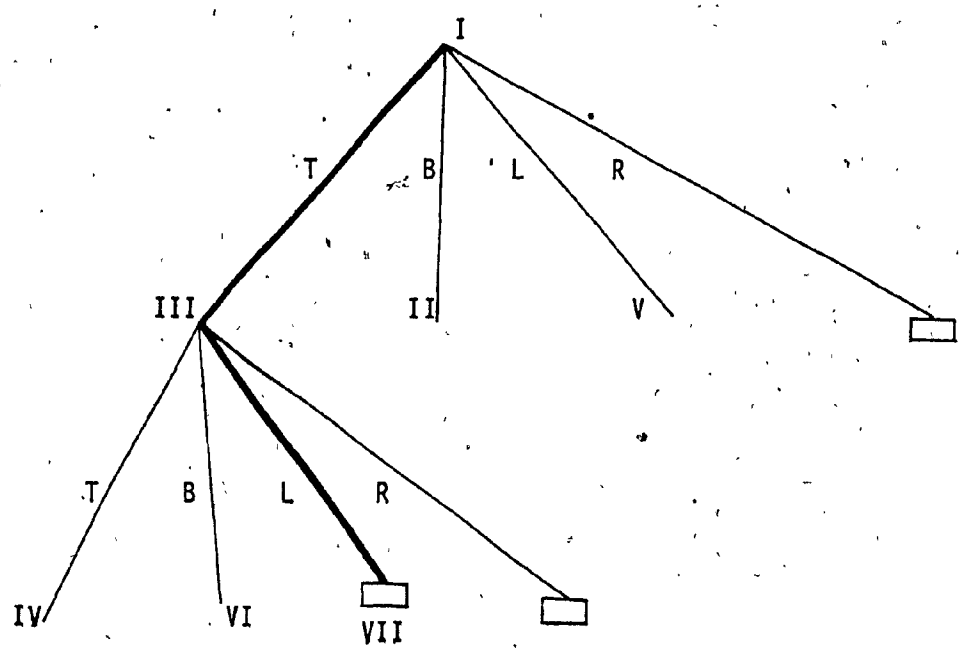
If $ymin(R_i) > ymax(R_0)$ then R_i is to the right of R_0

Note that at most two of these conditions can be true. For example R_i can be on top and to the right of R_0. For our purposes either condition could be used to determine the proper branch to take. Figure 3.8 shows one such insertion.

In addition, information must be stored in the tree about whether a rectangle extends partially beyond the bounds of another. The basis for the tree structure is the relationship described above; that is, whether one rectangle is

Figure 3.8 Inserting a rectangle into the quaternary tree.

rectangle	x min	y min	x max	y max
I	5	3	5	5
II	6	1	6	4
III	2	4	3	6
IV	1	5	1	6
V	3	1	5	1
VI	4	2	4	4
VII	1	1	2	3



The insertion of VII involves testing with I where it is found to be on top, and testing with III, where it is found to be to the left.

completely to the right, left, top or bottom of another. It is possible, for example, that a rectangle R_i which is completely to the left of another, R_0, could extend above and/or below R_0. This information will be useful in detecting overlap between new potential rectangles and already existing ones. These partial relationships are tested at each node encountered when inserting a new rectangle. Thus, at most $\log(x)$ such tests are required for insertion if the tree is height balanced. There are, again, four tests for partial relationships, although once the complete relationship is found at most two partial ones can exist, as illustrated above. The four additional tests are:

If $xmax(R_i) > xmax(R_0)$ then R_i extends below R_0

If $xmin(R_i) < xmin(R_0)$ then R_i extends above R_0

If $ymax(R_i) > ymax(R_0)$ then R_i extends to the right of R_0

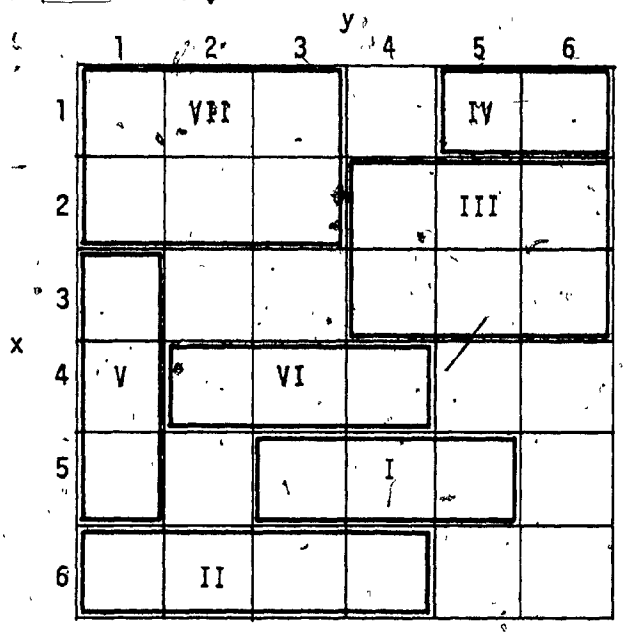
If $ymin(R_i) < ymin(R_i)$ then R_i extends to the left of R_0

An example including these relationships is given in Figure 3.9.

In order to ensure that the maximum search time is $O(\log x)$ the tree must be height balanced.

For quaternary trees we have adopted a modified definition of a balanced tree. We consider a quaternary tree

Sample Covering



The quaternary tree for this example including partial relationships. Partial relationships are shown in parentheses at the leaf nodes.

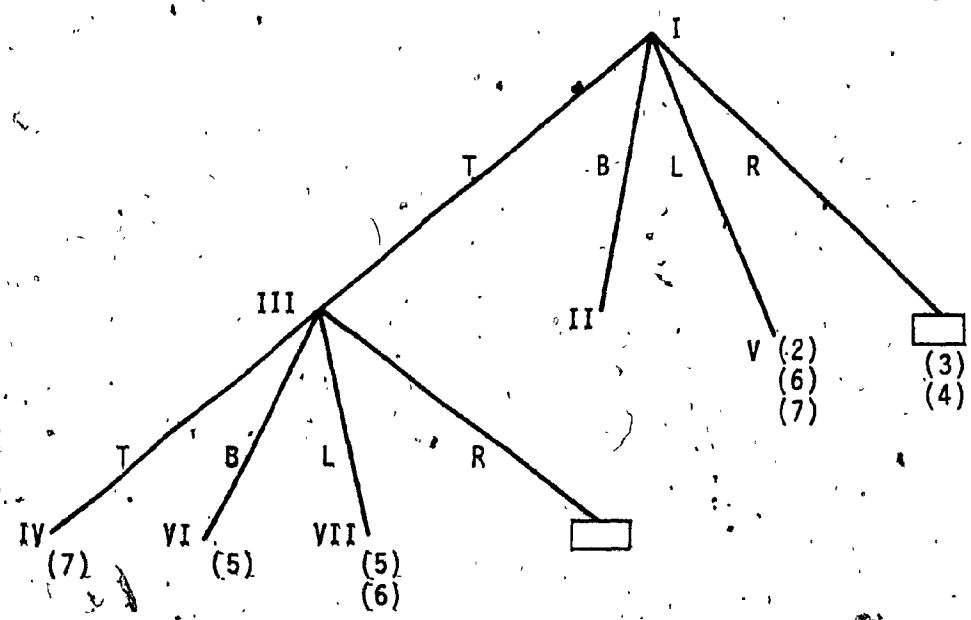
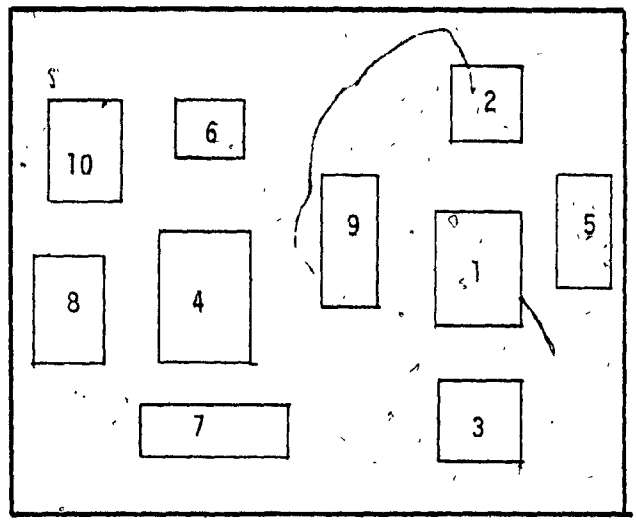


Figure 3.9

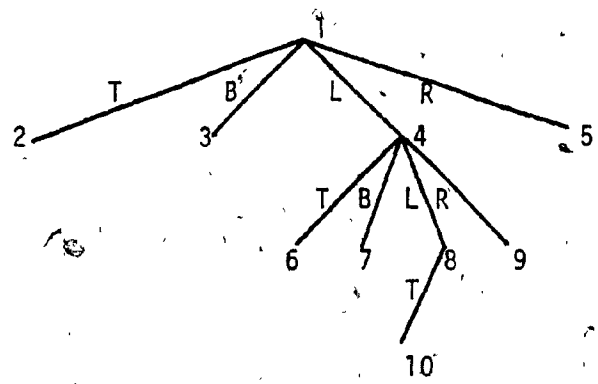
height balanced if, for every node, the height of the largest subtree differs from the height of the next largest subtree by no more than one. This assures that the overall height of the tree will be no more than $\log_2(i)$ where i is the number of internal nodes. This maximum height occurs if two sons are empty for each node. An example of an insertion requiring rebalancing is given in Figure 3.10. Further investigation reveals two types of rebalancing, analagous to the two cases for binary trees, one requiring a single rotation and the other a double. Figure 3.11 shows an insertion requiring a double rotation. This occurs whenever the insertion was in the left subtree of the right subtree of the unbalanced node (or vice versa), or was in the bottom subtree of the top subtree of the unbalanced node (or vice versa). The two general cases for balancing quaternary trees are given in Figures 3.12a and b.

We are now ready to analyze algorithm NEAROPTPART. Step 1 is executed exactly once. For each of the m vertices a pair can be formed with at most all of the $m-1$ vertices. Therefore the cost of this step is proportional to m^2. Steps 2 and 3 are performed once for each pair in the pairs lists and each require a constant cost. The total cost of these steps is also $O(m^2)$.

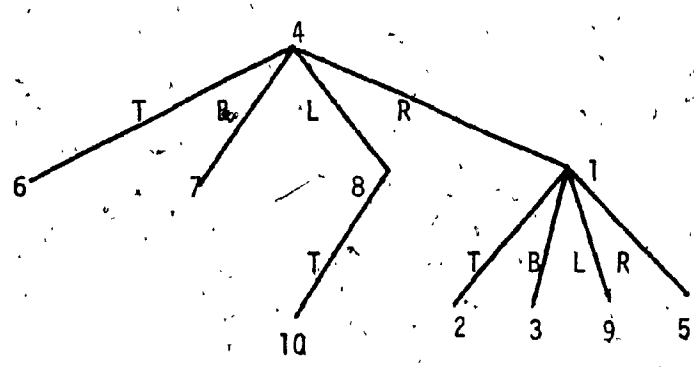
The creation of a new potential rectangle in Step 4 is done once for each rectangle eventually output (x) and once for each rectangle eventually aborted (z'). The cost of this step is a constant since it consists of marking two



a) The set of rectangles.

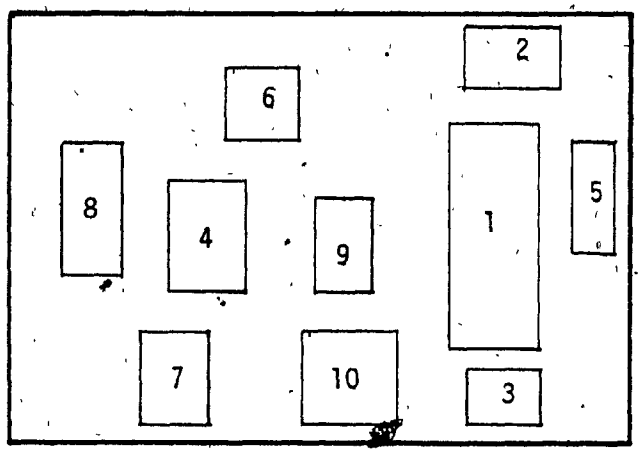


b) The insertion of 10 causes the root node to be unbalanced.

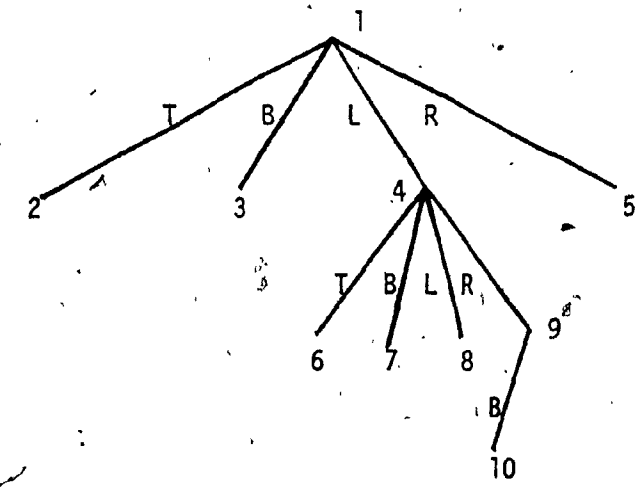


c) Tree after rebalancing - single rotation.

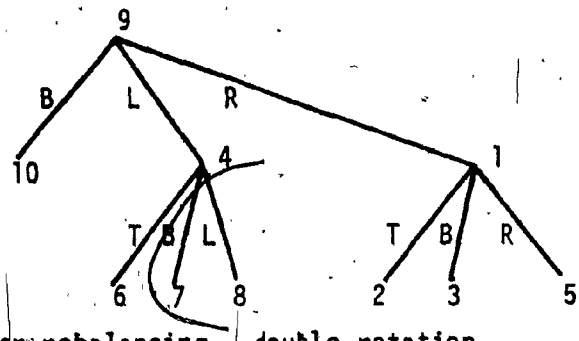
Figure 3.10 Insertion of a rectangle and rebalancing.



a) The set of rectangles.

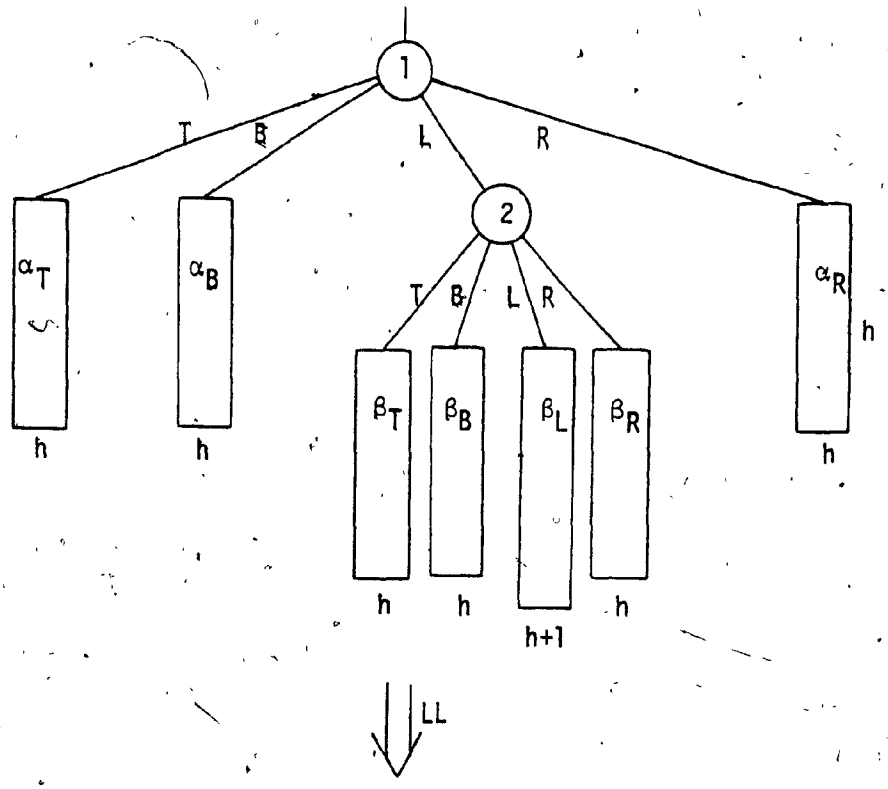


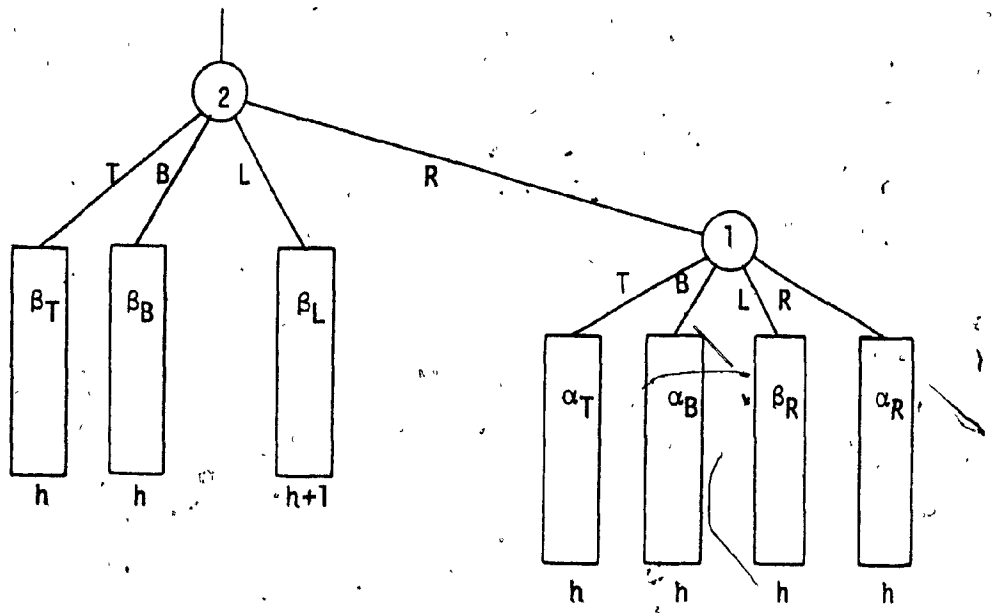
b) The insertion of 10 causes the root node to be unbalanced.



c) Tree after rebalancing - double rotation.

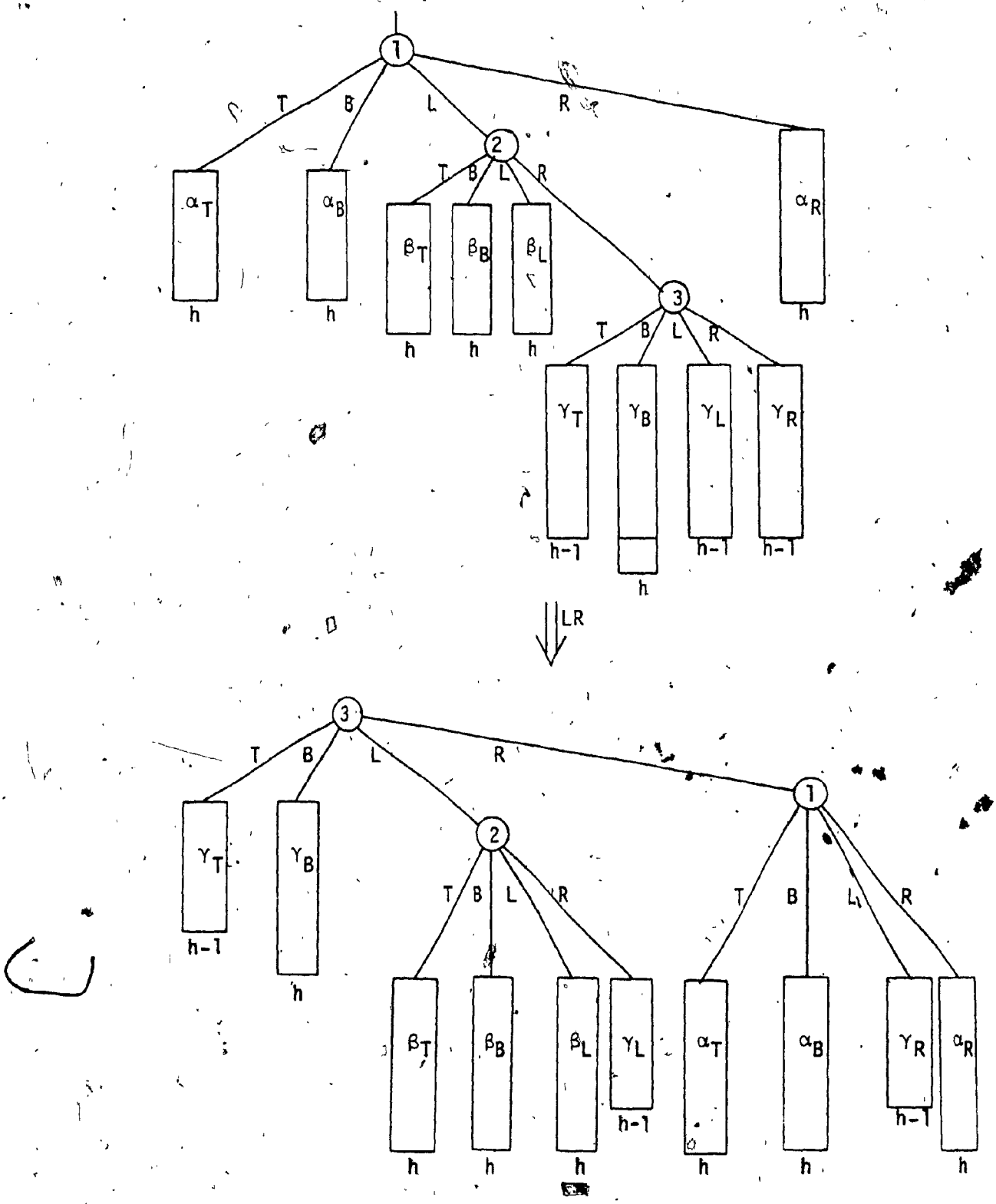
Figure 3.11 Insertion requiring a double rotation.





Case 1 - Single rotation. In this example the unbalanced subtree could have been β_T or β_B.

Figure 3.12a The two general cases for balancing quaternary trees.



Case 2 - Double rotation. In this example the unbalanced subtree could have been γ_T, γ_L or γ_R. Three other similar instances of double rotations can occur, namely RL, TB, and BT.

Figure 3.12b The two general cases for balancing quaternary trees.

vertices, adding their weights and entering the maximum and minimum row and column values of the vertices to the information about the rectangle. The maximum number of times this step can be executed is m, since $z' \leq z \leq y \leq m-x$ from relations r5, r4 and r3, so $z'+x \leq m-x+x = m$.

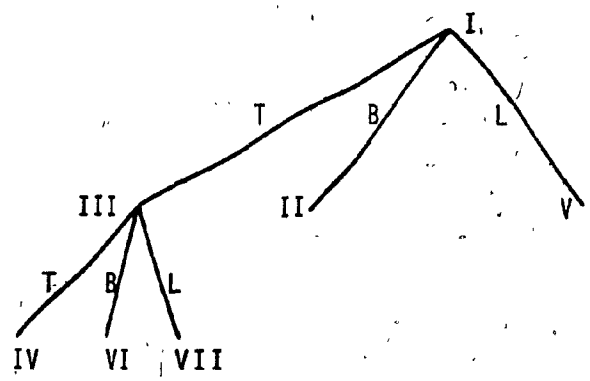
Steps 5 to 7 comprise the procedure for finding the next vertex to add to a rectangle once it has been started. Steps 5 and 7 require only a constant amount of time for testing the weight of the rectangle (step 5) and marking a vertex and updating the weight and limits of the rectangle if a new vertex is found (step 7). The work done in step 6 is as follows: Any unmarked vertex which appears in any of the adjacency lists of a vertex in the rectangle being formed must be considered. While this will be less than m, it is still of order m. For each of these eligible vertices, rectsize (or rectdist) and overlap must be computed. Rectsize takes a constant amount of time requiring just the updating of the limits of the rectangle and a simple calculation of its size. Similarly, rectdist requires calculating the rectilinear distance from the potential new vertex to all the vertices in the rectangle, of which there are less than the constant t or else the rectangle would be complete. Overlap, however, involves comparing the limits of the potential new rectangle with each of the previously formed rectangles, which could be as many as x. This search can be performed in $\log(x)$ time using the quaternary tree in which the rectangles are stored. The process is the same as

insertion of a new rectangle, except that if a rectangle is found for which the new potential rectangle is neither above, below, to the left or right, then these rectangles overlap. If a leaf node is found then no overlap existed and the new rectangle would be inserted there if it is complete. Figure 3.13 shows such a search, where overlap is detected. Thus, for each of the order m vertices considered in this step, $0(\log x)$ calculations are needed. By r1 and r2 the cost of step 6 becomes $0(N \log x)$ or $0(N \log N)$ since N is proportional to m and x.

The number of times steps 5 to 7 are performed is equal to the number of vertices in all the rectangles created (either finished or aborted) minus at least one vertex in each rectangle which was the initial vertex found in step 4. The total number of vertices used in all the completed rectangles, before left-overs are added at the end, is $m-y$. Subtracting one initial vertex for each rectangle gives $m-y-x$. In all of the z' aborted rectangles, there were z vertices, so $z-z'$ vertices at most were added in steps 5 to 7. Thus the total number of times those steps are executed is $m-y-x+z-z'$. Since $z \leq y$ from r4, this is less than m or N. Because step 6 has a cost of $N \log N$, however, and can be executed $0(N)$ times, it has a total cost of $N^2 \log N$ and is the most complex step in the algorithm.

It must be noted that, because of the partial relationships included in the quaternary tree, the search time may

rectangle	x min	y min	x max	y max
I	5	3	5	5
II	6	1	6	4
III	2	4	3	6
IV	1	5	1	6
V	3	1	5	1
VI	4	2	4	4
VII	1	1	2	3



Q: Does R = 1, 3, 3, 4 overlap?

Test with I: x max(R) < x min(I)? True - take branch T
Test with III: x max(R) < x min(III)? False
x min(R) > x max(III)? False
y max(R) < y min(III)? False
y min(R) > y max(III)? False
∴ overlap exists between R and III

Figure 3.13 Detecting overlap

not be $O(\log x)$ in all cases. Indeed, Figure 3.14 gives a "worst case" example where the data is skewed and the number of additional comparisons needed at some leaf nodes raises the search time to $O(x)$. This makes the cost of this step $O(Nx)$ or $O(x^2)$ by relation r2, and the overall cost of the algorithm $O(x^3)$ for this worst case. Our results on random input data indicate, however, that this situation is quite rare.

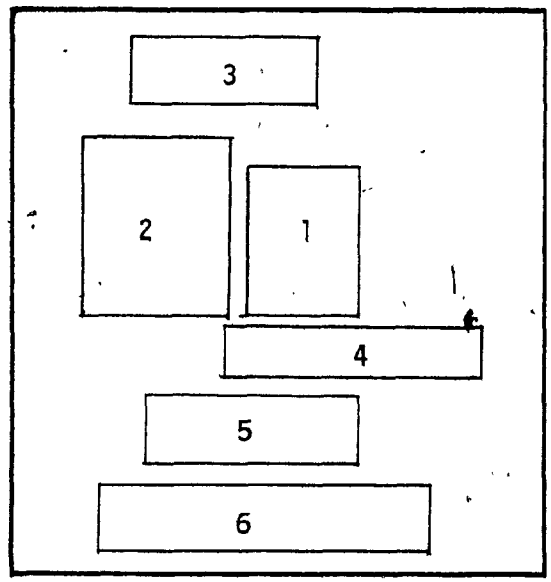
Finally, step 8 is performed z' times at a constant cost and step 9 requires calculating the distance between each of the y left-over vertices and each of the x rectangles. The cost of this step, then, is $x \cdot y$ which by r2 and r3 is less than or equal to $(m-x) \cdot (N/t) \leq N^2$.

3.7.3 Conclusions

The cost of partitioning a database using the algorithms FINDADJLIST and NEAROPTPART has been shown to be $O(D^2)$ for Phase I and $O(N^2 \log N)$ or $O(x^2 \log x)$ in most cases for Phase II. This results in an overall cost of $O(D^2 + N^2 \log N)$. We can easily see that, in all but the very sparse cases where $N << D$, the overall cost will be bounded by the Phase II cost. Otherwise, scanning the matrix in Phase I will be the greater cost.

3.8 Performance of the algorithm

We have tested the algorithm on randomly generated databases of several different sizes, varying the value of the threshold, t, and the sizes of the domains D_1 and



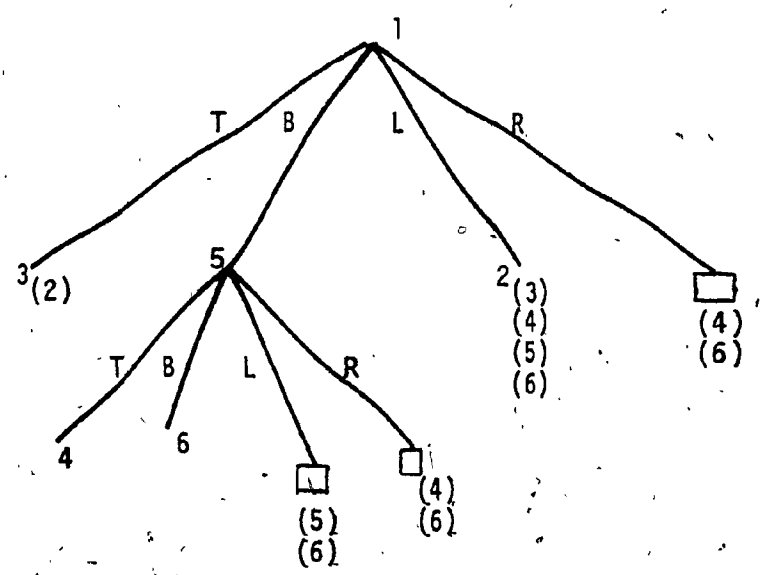


Figure 3.14 A Worst case for quaternary tree search.

D_2. The results show that our algorithm is indeed producing a nearly optimal partitioning, producing approximately 80% of the maximum number of regions possible for the various values of N and t. The total area covered by the rectangles is about 70% of the matrix for densities of less than 1.0. Two examples of the partitioning produced by our algorithm are given in Figure 3.15.

By way of comparison, Figure 3.16 presents three different partitions of one sample database. The figure shows Yu and Chin's method, our method and an optimal partioning for this database. The results of these methods are summarized in Figure 3.17. In over 50 test cases, none were found to exhibit the property of having an excessive number of partial relations in the leaf nodes of the quaternary tree, supporting our contention that this is a rare phenomenon.

3.9 Extending the algorithm to higher dimensions

We have developed a scheme whereby databases with more than two domains can be split into disjoint partitions. The first step in this procedure is to partition the database on the basis of the first two domains only. This yields a set of rectangles which can now be used in further applications of the algorithm. For all the remaining domains, we set up a new matrix with the rectangles previously found as one axis and the next domain as the other. All the records can be plotted on this new matrix as they all belong to some rectangle and have some value in the next domain. The

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

a) Ex. 1: $N = 50$, $t = 5$, $X = 9$ $D_1 = 5$, $D_2 = 10$

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

b) Ex. 2: $N = 50$, $t = 3$, $X = 14$ $D_1 = 10$, $D_2 = 10$

Figure 3.15 Examples of partitioning using random data points.

	1	2	3	4	5	6
1	X				X X	X X
2	X		X	X X	X	
3	X				X	X
4	X	X	X	X		
5	X		X		X X	
6	X	X X		X		

a) unpartitioned database

	1	2	3	4	5	6
1	X				X X	X X
2	X		X	X X	X	
3	X				X	X
4	X	X	X	X		
5	X		X		X X	
6	X	X X		X		

b) Yu and Chin's method

	1	2	3	4	5	6
1	X				X X	X X
2	X		X	X X	X	
3	X				X	X
4	X	X	X	X		
5	X		X		X X	
6	X	X X		X		

c) our method

	1	2	3	4	5	6
1	X				X X	X X
2	X		X	X X	X	
3	X				X	X
4	X	X	X	X		
5	X		X		X X	
6	X	X X		X		

d) optimal partitioning

The thick lines indicate the boundaries of the partition.

Figure 3.16 Comparison of three methods for partitioning a sample database.

	d_1	d_2	N	t	x	N/x	N/d_1d_2	area covered d_1d_2	x/N/t
Yu & Chin's method	6	6	25	3	5	5.0	.694	91.7%	62.5%
our method	6	6	25	3	7	3.57	.694	75%	87.5%
optimal method	6	6	25	3	8	3.125	.694	63.9%	100%

Figure 3.17 Comparison of the Performance of Three Methods

algorithm then partitions this new matrix, producing a new set of regions involving the added domain. The process is repeated until all the domains have been used. Hence, if the dimensionality of the database is k, the algorithm is applied $k-1$ times.

To illustrate this procedure, a sample database with three domains is given in Figure 3.18, and the rectangles produced using the first two domains are shown in Figure 3.19. Figure 3.20 gives the new matrix with the third domain as columns and the rectangles as rows, along with the regions produced.

The cost of each iteration is still $0(N^2 \log N)$ since there are still at most n nonempty cells in each matrix although the size of the matrix and the distribution of the vertices will vary with each pass. Therefore the overall cost for partitioning a database with k domains is proportional to $(k-1)(N^2 \log N)$.

We have implemented several variations of this method for $k=4$. The first performs two separate partitionings, one using the first two domains and the other the last two. This results in two sets of rectangles which are then each considered as an axis of a third matrix, and this is partitioned to give the final set of regions. Note that the same number of iterations are still needed for this method. This will be true for any k. For example, consider $k=16$. If we do eight partitionings using two domains at a time, we must do four more using two of those sets of rectangles each, then

record #	D1	D2	D3		record #	D1	D2	D3
1	1	1	1		14	4	1	2
2	1	5	1		15	4	2	2
3	1	5	3		16	4	3	4
4	1	6	2		17	4	4	2
5	1	6	3		18	5	1	3
6	2	1	1		19	5	3	1
7	2	3	4		20	5	5	3
8	2	4	2		21	5	5	4
9	2	4	3		22	6	1	1
10	2	5	2		23	6	2	3
11	3	1	1		24	6	2	4
12	3	5	2		25	6	4	4
13	3	6	2					

Figure 3.18 Sample database with three domains.

	1	2	3	4	5	6
1	X	1			XX	2 XX
2	X		X	XX	X	3
3	4 X				X	X
4	X	5 X	X	X		
5	X		6 X		XX	
6	X	7 XX		X		

rectangle 1: $1 \leq D_1 \leq 2$ $1 \leq D_2 \leq 3$

rectangle 2: $1 \leq D_1 \leq 1$ $5 \leq D_2 \leq 6$

rectangle 3: $2 \leq D_1 \leq 3$ $4 \leq D_2 \leq 6$

rectangle 4: $3 \leq D_1 \leq 5$ $1 \leq D_2 \leq 1$

rectangle 5: $4 \leq D_1 \leq 4$ $2 \leq D_2 \leq 4$

rectangle 6: $5 \leq D_1 \leq 5$ $3 \leq D_2 \leq 5$

rectangle 7: $6 \leq D_1 \leq 6$ $1 \leq D_2 \leq 4$

Figure 3.19 Rectangles produced by first two domains.

	1	2	3	4
R1	1 XX			3 X
R2	X	2 X	XX	
R3		XX 4 X	X 5	X
R4	X 6	X 7	X	
R5		XX		X
R6	X		X	X 8
R7	X		X	XX

Cubic regions produced:

1. $1 \leq D_1 \leq 2$ $1 \leq D_2 \leq 6$ $D_3 = 1$
2. $1 \leq D_1 \leq 1$ $5 \leq D_2 \leq 6$ $2 \leq D_3 \leq 3$
3. $1 \leq D_1 \leq 5$ $1 \leq D_2 \leq 6$ $D_3 = 4$
4. $2 \leq D_1 \leq 3$ $4 \leq D_2 \leq 6$ $D_3 = 2$
5. $2 \leq D_1 \leq 6$ $1 \leq D_2 \leq 6$ $D_3 = 3$
6. $3 \leq D_1 \leq 6$ $1 \leq D_2 \leq 5$ $D_3 = 1$
7. $3 \leq D_1 \leq 5$ $1 \leq D_2 \leq 4$ $D_3 = 2$
8. $5 \leq D_1 \leq 6$ $1 \leq D_2 \leq 5$ $D_3 = 4$

Figure 3.20 New matrix with third domain.

two more iterations and finally one last partitioning for a total of fifteen.

Another variation combines the values of two domains to make one larger domain. In this way, four domains can be used in one two-dimensional matrix. The method is simple; if one domain has D_1 different possible values and another has D_2, we create a new domain with $D_1 * D_2$ values, those being all the combinations of values, one of which comes from the first domain and the other from the second. Thus if $D_1 = 3$ and $D_2 = 2$ the new domain consists of (1,1)(1,2)(2,1)(2,2)(3.1) and (3,2). We can now set up a matrix which has two combined attributes for one axis and the other two for the second axis and perform a single iteration of the algorithm to produce the final partitioning. The drawback of this method is that adjacent values of some attribute might not be in adjacent rows or columns. Since our aim is to create partitions which contain records that are as similar as possible, this method is less desirable theoretically.

A final variation concerned the order in which the domains were considered in our original method, whether it be by decreasing or increasing size of the domain.

Our results showed that none of the variations produced a better partitioning than our original method, although our tests were by no means exhaustive. The only variation which may require less work than the other methods is combining values. However, since we have theor-

etical misgivings about the partitions produced in this manner, we have chosen to use the original method proposed in this section for partitioning databases with greater than two domains.

CHAPTER FOUR

HIERARCHICAL PARTITIONING

4.1 Introduction

The partitioning algorithm presented in the previous chapter maps the database onto a matrix and builds partitions by combining neighbouring cells until they attain a certain size. It is essentially a "bottom-up" or merging process, starting with many small groups and creating fewer larger groups. In this chapter we present a "top-down" technique which starts with the entire database as one group and successively splits it up into smaller partitions. The process can be represented by a tree and, in fact, is implemented as a tree. Each node contains a number of records and a node q is split on an attribute A_i by creating one child node for each possible value of A_i and allocating every record at q to a child according to the value of A_i in the record. A node is split only if all its children can have at least t records.

It is clear that this method is fundamentally different from the merging algorithm discussed in the last chapter. Thus the type of partitions produced will also be different in terms of the relationship of the records to each other. We will address this matter more fully in section 4.5 as well as give the results of algorithms using the top-down technique. Section 4.2 presents the algorithm, its refinements and variations. The formal algorithm is given in

section 4.3, and the cost is analyzed in section 4.4.

4.2 Informal description of the algorithm

The top-down partitioning algorithm starts with one node containing all the records in the database. These records are then split on the basis of their values for some attribute. Hence, the number of children created is the number of different possible values, or size of that attribute. If, after such a splitting, any of the children nodes contain fewer than the threshold t records, all the records are returned to the parent node and the splitting is attempted using another attribute. If the splitting is successful, the same process is applied to all the children. The algorithm proceeds recursively to split every node until each attribute has been used, either successfully or not, in any given path from the root to a leaf node.

Let us consider a small example of partitioning using this technique. Assume the database consists of fifty records and there are three attributes, A_1 to A_3. We will indicate the size of an attribute A_i by d_i. In our example, let $d_1 = 5$, $d_2 = 3$ and $d_3 = 2$. This means there are five possible values for A_1, three for A_2 and only two for A_3. The threshold, t, will be set to 3. Figure 4.1 gives a sample database with these parameters.

The first step is to construct a root node containing all 50 records. Call this node R_1. We will denote the number of records at a node P as $|P|$; therefore $|R_1| = 50$. Now we split R_1 on the basis of attribute A_1,

	A_1	A_2	A_3		A_1	A_2	A_3
1.	1	1	1	26.	3	2	1
2.	1	1	2	27.	3	2	1
3.	1	1	2	28.	3	2	2
4.	1	2	1	29.	3	3	2
5.	1	2	2	30.	3	3	1
6.	1	2	1	31.	3	3	2
7.	I	2	2	32.	4	1	1
8.	1	3	2	33.	4	1	2
9.	1	3	1	34.	4	2	2
10.	1	3	2	35.	4	2	1
11.	2	1	1	36.	4	3	2
12.	2	1	1	37.	4	3	1
13.	2	1	1	38.	4	3	2
14.	2	1	2	39.	5	1	1
15.	2	1	2	40.	5	1	2
16.	2	1	2	41.	5	1	2
17.	2	2	1	42.	5	1	1
18.	2	2	2	43.	5	1	1
19.	2	2	1	44.	5	2	2
20.	2	3	2	45.	5	2	2
21.	2	3	1	46.	5	2	1
22.	2	3	2	47.	5	3	1
23.	3	1	1	48.	5	3	2
24.	3	1	2	49.	5	3	1
25.	3	2	2	50.	5	3	2

Figure 4.1 Sample database with 50 records and three domains.

thus creating five children since $d_1 = 5$. According to our sample database the five children nodes, S_1 to S_5, have the following number of records: $|S_1| = 10$, $|S_2| = 12$, $|S_3| = 9$, $|S_4| = 7$ and $|S_5| = 12$. Next we consider S_1 and try to split on attribute A_2. The splitting is successful, creating nodes T_1, T_2 and T_3 with sizes 3, 4 and 3 respectively. Obviously, none of these nodes can be split further if each child is to have at least 3 records. These three nodes, then, are leaf nodes of our partitioning tree and each represents a partition of the database. The next node to be processed in this recursive procedure is S_2 which has 12 records. It is successfully split on attribute A_2, yielding nodes T_4, T_5 and T_6. Node T_4, which contains six records, can be further split on attribute A_3 producing two more nodes U_1 and U_2 with three records in each. Note that all three attributes have been used along the path from the root R_1 to U_1 and U_2 and each record in U_1 has values 2 for A_1, 1 for A_2 and 1 for A_3. Similarly all the records in U_2 have values 2 for A_1, 1 for A_2 and 2 for A_3. Although we can attempt to split T_5 and T_6 on attribute A_3 as well, it will be unsuccessful and they remain as leaf nodes. Our attention is now turned to node S_3 which contains nine records. The attempt to split on A_2 fails at this node since only two of its records have the value 1 for A_2. The records, then, are returned to S_3. We try to split using attribute A_3. This is successful and the two nodes T_7 and T_8 are

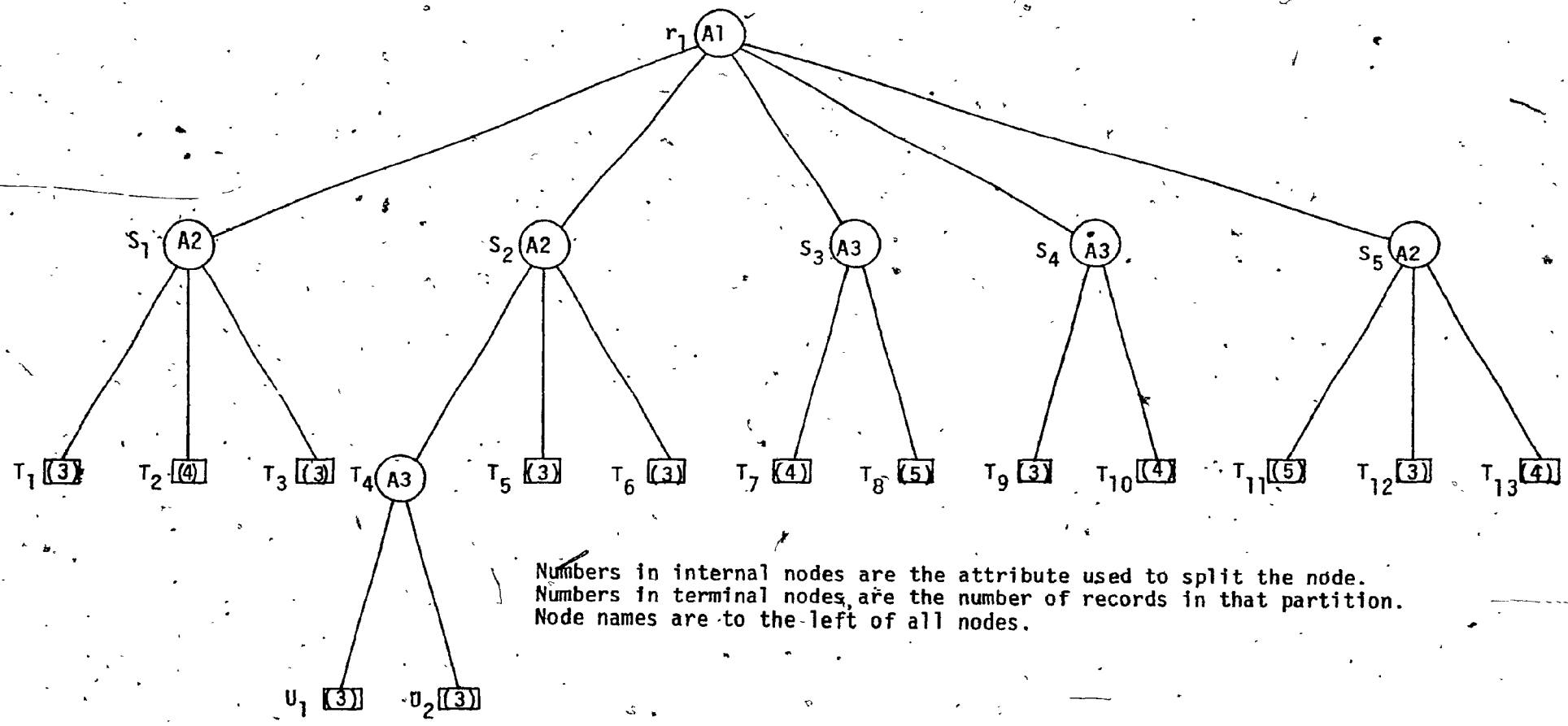
created containing 4 and 5 records each. No further partitioning can be attempted for these nodes as all the attributes have been considered, if not used, in forming the nodes. Finally, nodes S_4 and S_5 are split in the same manner. Figure 4.2 shows the entire partitioning tree for this example.

4.2.1 The hierarchy of attributes used for splitting

The order in which the attributes are used to split the nodes in the partitioning tree is of critical importance. First, the higher an attribute is in the order, the greater chance it has of successfully splitting a node. This is true because initially there are more records in a node. As one goes further down the tree, there are fewer records per node and the chance of splitting for a given size d_i of an attribute A_i is less. Secondly, we would like to arrange the attributes in an order which will provide for the greatest number of attributes to be used in the tree. The records in a partition created by this method are related to each other (and, indeed, all have the same values) only by the attributes used along the path from the root to the partition. They are not likely to have the same values for the other attributes. Thus a query specifying an attribute which has not been used to form any partitions may involve some of the records in many different partitions, giving less accurate results.

The question is, what is the ordering which will involve the greatest number of attributes? It turns out

Figure 4.2 Partitioning tree for the sample database.



Numbers in internal nodes are the attribute used to split the node.
Numbers in terminal nodes are the number of records in that partition.
Node names are to the left of all nodes.

that if we order the attributes by decreasing size, i.e., $d_1 \geq d_2 \geq \ldots d_K$, the number of attributes used will be maximized. By this method the largest nodes, which occur in the highest levels of the tree, are split the greatest number of ways possible. Although the number of records per node decreases rapidly, we attempt to split the smaller nodes at lower levels in fewer ways, and there is a better chance those splits will be successful than if we used the reverse ordering. Also, if a node cannot be split on, say attribute A_i, it is more likely to be split on A_{i+1} since this attribute has fewer values, whereas if we ordered the attributes by increasing size, it would become increasingly harder to split nodes as we go down the tree and less likely as we go through the attributes.

Regarding the first consideration of ordering attributes (that the first few attributes in the ordering are more likely to be successfuly used) we could take into account the query probabilities of each attribute. We define an attribute's query probability as the probability that a query will specify a value (or values) in that attribute. If we have statistics on past querying patterns, we can calculate these probabilities and order the attributes by descending query probability. Thus the most queried attribute will be used first and the least queried attribute will be used last and be the least likely to be used in creating a partition. The queries involving the rarely queried attributes will be less accurate, but will not be

asked often and the overall accuracy of the statistics will not be greatly affected by those queries.

Ordering the attributes solely on the basis of query probability suffers from the drawback of not taking the size of the attributes into account. This may cause fewer attributes to be used altogether even though the attributes used in splitting are most probable. It is possible to use a combination of the two methods in the following way: define a cut-off for query probabilities; call this p_{min}. For all the attributes with probability greater than p_{min}, order them by decreasing size, and take these attributes first. Then order all the other attributes by descending size and use them after the first group. More formally, suppose j of the k attribute have query probabilities $p \geq p_{min}$. Call these $A_1 \ldots A_j$ and the others $A_{j+1} \ldots A_K$. The order of all the attributes will be:

$$A_1, A_2 \ldots A_j, A_{j+1} \ldots A_k \text{ where}$$

$$d_1 \geq d_2 \geq \ldots d_j \quad \text{and} \quad d_{j+1} \geq d_{j+2} \geq \ldots d_K .$$

Also, $p_i \geq p_{min}$ for $1 \leq i \leq j$ and $p_i < p_{min}$ for $j < i \leq K$. This scheme allows for the maximum number of attributes to be used for partitioning with the most likely queried attributes used first. We will discuss the statistics produced by these different variations on the top-down partitioning algorithm in section 5.

4.2.2. Refinements of the algorithm

The partitioning algorithm described above may produce some partitions with many more than t records in them. This happens because, unlike the bottom-up rectangular partitioning method whose regions are merged until they contain at least t records, the top-down technique starts with large partitions and tries to split them into smaller ones. Sometimes it will not be possible to split such a partition due to a skewed distribution of the records' values for some attribute. For this reason, we have devised two more passes which will decrease if not totally eliminate the chances of creating partitions with large numbers of records. We have defined "large" in this context to be greater than or equal to twice the threshold t.

The second pass simply groups together all the records from partitions containing at least $2t$ records, and re-applies the algorithm to those records. The order of the attributes may be slightly different for this pass since we will take any unused attributes first, in the order previously determined, then the other attributes in order. This, and the fact that we are starting with a different group of records than in the first pass, results in a different splitting and different partitions from the second pass in most cases. It is possible, however, that some of the partitions remain the same, and it is likely that some of the new partitions have more than $2t$ records. These partitions are the ones examined in the third pass.

After two attempts to split the database into small partitions by the above method, we are now ready to examine more closely the large partitions that still remain and apply a heuristic method to finally break them up.

The third pass works separately on each large group remaining after the second pass. The distribution of all the records in such a group is obtained according to their values for each attribute. Then the records with adjacent values are grouped together if need be to create possible partitions. This is done for all the attributes and the one which provides the best partitioning is chosen.

Consider an example where again we have three attributes with sizes $d_1 = 5$, $d_2 = 3$ and $d_3 = 2$. Suppose there are nine records in a leaf node of the tree after the second pass. The distribution of these records by values for each attribute is shown in Figure 4.3. Let us examine these distributions and form partitions according to our heuristic. There are four records with value 1 in the first attribute. These can form one partition. The three records with values 2 or 3 can also form a partition, but the two records left out cannot, so they must be included in the previous partition. Thus we can form two partitions on attribute A_1, one containing the records with value 1 and the other consisting of all records with values 2 to 5. We now look at the second attribute. Only two records have value 1 so these must be combined with the records having the next value, 2. There are five of these,

rec#	A_1	A_2	A_3
1.	1	1	1
2.	5	2	1
3.	2	2	1
4.	1	3	2
5.	3	2	1
6.	1	2	1
7.	3	1	1
8.	5	3	2
9.	1	2	1

a) Nine records

Attribute	1	2	3	4	5
A_1	4	1	2	0	2
A_2	2	5	2		
A_3	7	2			

b) Distribution of the records by values

Figure 4.3 Third pass applied to nine records.

making a partition of seven records. Furthermore, there are only two records with value 3 and these must be included in the one partition possible with this attribute. An important aspect of our heuristic is illustrated here. It would have been possible to make two partitions, one consisting of the five records with value 2 and the other of the four records with values 1 or 3. To search for combinations of values in this way is a complicated procedure, however, and we feel that it is not worth the increased amount of time required. It would not in this case, for instance, find a better partitioning than that found for A_1. Considering that two attempts have already been made to create as small partitions as possible, we prefer a simple heuristic which simply scans the distributions one value at a time and combines adjacent values if necessary. This method is also preferable in terms of the relationship of records within a partition. It is better to have partitions in each of which records have adjacent values than widely dispersed ones.

Finally, a scan of the distributions for attribute A_3 yields only one possible partition. Since the records can be split into the greatest number of partitions using attribute A_1, we choose that splitting for this example.

All leaf nodes from the second pass with at least $2t$ records will be processed in this manner. The second and third scans combine to improve the total number of partitions and limit the number of large partitions, sometimes dramatically. Some results on these improvements will

be given later in this chapter.

4.3 Formal description of the algorithm

The top-down tree splitting algorithm is presented formally below. It includes the second and third passes. Assume we have N records and K attributes whose sizes are given by $d_1 \ldots d_K$. Each node contains a list of the records at that node, a field indicating the number of those records, and pointers to as many as d_{max} children, where d_{max} is a constant indicating the maximum number of values for any attribute. A stack is used which stores pairs (Q,i), where Q points to a node and i contains an attribute number. The variable "z" is used to indicate the current attribute number, "P" is a pointer to the current node, and "Pass" indicates the pass number (1, 2 or 3).

Algorithm TOPDOWNPART

[sort attributes] 1. Order the attributes $A_1, A_2 \ldots A_K$ such that $d_{A_1} \geq d_{A_2} \geq \ldots d_{A_K}$.

[initialize] 2. Create a node containing all N records. Call this P. Set $z \leftarrow 1$, pass $\leftarrow 1$.

[split] 3. Create d_{A_z} sons of node P and distribute the records at P among the sons according to the value of attribute A_z in the records.

[test split] 4. If each son has at least t records, go to step 6.

[invalid split] 5. Return the records in all sons to P. If $z < K$, set $z \leftarrow z + 1$ and return to step 3,
Otherwise go to step 7.

[valid split] 6. If $z < K$ then for each son S_i of P, $1 \leq i \leq d_{A_z}$, push the pair $(S_i, z+1)$, onto the stack.

[pop stack] 7. If stack not empty, pop a pair (Q,i) from the stack, set $P \leftarrow Q$, $z \leftarrow i$, and go back to step 3.

8. pass ← pass + 1

9. If pass > 2, go to step 13.

[second pass] 10. Create a node containing all the records from leaf nodes having $\geq 2t$ records. Set P to this node.

11. Redefine $A_1 \ldots A_K$ such that any unused attributes occur before used ones.

12. Set $z \leftarrow 1$ and go back to step 3.

[third pass] 13. Set P to the next leaf node from the second pass tree having $\geq 2t$ records. If no more exist, terminate.

14. For each attribute, plot the distribution of the records at P by values.

15. Form as many partitions as possible for each attribute.

16. Select the attribute which gives the best partitioning and split P accordingly.

17. Return to step 13.

4.4 Cost analysis

Here we present an analysis of algorithm TOPDOWNPART and obtain a cost function. The analysis is straightforward and is based largely on the partitioning tree structure, rather than a step by step analysis of the formal algorithm.

Let us consider the tree produced by the first pass of the algorithm. The root node contains all the records before it is split into some number of sons. In order to perform this splitting each record has to be examined once to determine which son it belongs to. Hence the amount of work done at this level is proportional to N. Before splitting, the total number of records contained in nodes at the second level is also N. Therefore the amount of work needed to split all the nodes on the second level, regardless of which nodes are split on which attributes, is also proportional to N since all N records must be examined once to determine which third level node they belong to. The same will be true for each subsequent level until leaf nodes are created.

Suppose all the splittings are successful for every node in the tree. Then the tree is of maximal size and all the leaf nodes are in level $K + 1$. Each record has undergone K splittings, once for each attribute. The cost of creating the partitions is $0(N)$ for each of the K levels at which nodes are split, giving a total cost of $0(KN)$ if each splitting is successful.

What if some of the splittings are unsuccessful? An unsuccessful split still examines all the records in a node before determining that the split is invalid. It costs the same to do an unsuccessful split as a successful one. If an invalid split occurs, however, all descendant leaf nodes of that node will be at least one level higher. The crux of the matter is this: a record will be examined for the purpose of splitting once for each attribute in the database. Whether or not a valid split results from such an examination is irrelevant. This can be discerned in the formal algorithm from the fact that the terminating condition in step 5 (for an invalid split) and step 6 (for a valid split) is $a = K$. Therefore the cost of partitioning the database in the first pass is $O(KN)$.

There is also a minor cost in step 1 to sort the attributes. This can be done in $O(K \log K)$ time. Since $K < N$ for any but the most unusual cases, $\text{Log}\, K << N$ and the cost of this step does not increase the overall cost of the algorithm.

The second pass is merely a reapplication of the first. It will usually involve fewer than N records, but the overall cost will be the same as that of the first pass, namely $O(KN)$.

Finally, the third pass involves scanning the K fields in each record considered in order to plot the distribution of the records by value for all attributes. Again, while this will involve less than N records most of the

time, the cost is still $0(kN)$ since the number of records involved in the third pass cannot be known to be much less than N and can actually equal N in the worst case. Once this step is performed we must look at the counts for each value of each attribute and find the best partitioning. This requires $0(k \cdot dmax)$ calculations for each partition with at least $2t$ records, where $dmax$ is a constant referring to the maximum number of values for an attribute. The number of such partitions is less than $N/2t$. Therefore the total cost for finding the best partitioning in the third pass (steps 14 to 16 in the algorithm) is $0(k \cdot dmax \cdot N/2t)$ or $0(kN)$ since $dmax$ and t are both constants. This makes the cost of the third pass also $0(kN)$ because plotting the distributions (step 13) has the same cost.

We have now shown that each pass of the algorithm has complexity $0(kN)$ which is therefore the overall cost of the algorithm TOPDOWNPART.

4.5 Performance of the algorithm

Unlike the rectangular partitioning algorithm, the types of groups formed by top-down splitting, at least in the first pass, are dependent on the density of the database. We define density to be the number of actual records divided by the number of possible different records, or $N/\prod_{i=1}^{k} d_i$.

If, for example, $N = 50$ and $k = 3$ with $d_1 = 5$, $d_2 = 3$ and $d_3 = 2$, the density is $50/(5 \cdot 3 \cdot 2)$ or 1.67. We

will arbitrarily call any database such as this with density greater than one, a dense database. The denser the database, the more attributes will be involved in splitting. The reason for this is fairly obvious. With the restriction that each partition must have at least 't' records, the density must be more than t for the possibility to exist that each partition is split on all the attributes. For densities less than t, some partitions will not be split on some attributes. As the density decreases, some attributes may not be used at all. When these attributes are specified in a query, the records with a given value for an unused attribute will be in many different partitions and, worse yet, any partition may contain records with several different values for such an attribute. We would expect the statistics produced by sparser databases to be worse than those for denser ones, but only for the attributes not used in splitting. It is for this reason that we have implemented a variation of the algorithm using probabilistic queries, as introduced previously. Preliminary results confirm our earlier remarks, and the relative errors resulting from probabalistic queries are less than those using random queries for the same database. All the statistical results will be discussed in detail in section 5.

The effects of density on the number of partitions produced is less clear, but also seem to favor denser databases. As dense databases tend to involve more attributes, there is more chance of a large group being split into

smaller ones. Moreover, the larger the group, the better are its chances of being split, and denser databases have larger groups. Very sparse databases may produce poor splits because of the small groups produced when splitting and the greater chance of a split being unsuccessful. On the other hand, it may happen that a relatively sparse database produces a good splitting because of the very fact that smaller groups are made.

Most of these discrepancies are eliminated by the second and third passes. In some cases the improvement is dramatic, more than doubling the original number of partitions. The number of partitions produced after application of all three passes is approximately 70% of the total possible in all our test results. Table 4.1 gives some sample results including second and third pass improvements and comparisons with the rectangular merging algorithm.

We will conclude this section with a brief discussion of the differences between the merging algorithm and the splitting algorithm.

As can be seen from Table 4.1, the merging algorithm usually produces more partitions for a given database. The reason for this lies in the fundamental difference between the algorithms. The merging procedure starts with small groups, usually less than t records, and adds records until there are just t records or more in the group. Before left-overs are added at the end, partitions formed in this manner rarely have more than t records. Thus the number of

Table 4.1 Number of Partitions Formed by Both Methods

		Merging alg.		Splitting algorithm		
N	t	#	% of opt.	# after pass 1	final #	%
100	3	30	91%	20	27	82%
100	5	17	85%	14	14	70%
500	3	137	82.5%	87	124	75%
500	5	81	81%	61	70	70%
1000	3	278	83%	152	224	67%
1000	5	171	85%	71	143	71%

partitions is close to maximum. This is not the case for the splitting algorithm, which starts with one large group and successively makes smaller and smaller ones. We have added some refinements to this method in order to prevent very large groups from existing in the final partitioning, but that cannot insure as many partitions as the merging process produces.

One major advantage of the splitting algorithm is in the composition of the partitions. It has always been a criticism of the partitioning method for protecting a statistical database that the groups may be ill-defined and the statistics released therein would be meaningless. The groups formed by our top-down algorithm are very homogeneous. All the records in a partition produced by the first or second pass have exactly the same values for each attribute on which they were split. Groups made by the third pass will have the same values for all the attributes used in splitting them on the second pass, and adjacent values for the attribute found in the third pass. This is not the case for the merging algorithm, although we have tried to make the groups as homogeneous as possible. The rectangles formed by the first algorithm represent ranges of values for the attributes used as opposed to specific values. Moreover, when the algorithm is reapplied for higher dimensions the narrow ranges made initially may become expanded (see Figures 3.19 and 3.20). We expect, then, that the statistics produced by the splitting method will be as good as those

from the merging procedure, even though fewer partitions are made.

Another advantage of the splitting algorithm is simplicity and therefore speed. It is much less complex than the region forming algorithm, having cost $0(KN)$ as opposed to $0(KN^2 \log N)$. Actual implementation of the algorithms shows that the splitting algorithm does indeed run in time linearly proportional to N, whereas the merging algorithm uses time proportional to N^2. This difference becomes rather substantial when N is large as will be shown in the next chapter.

CHAPTER 5

RESULTS

5.1 Introduction

In this section we will describe the querying system which we have implemented for statistical databases, and the results of the experimental trials we have run.

It should be emphasized that the query answering system is independent of the partitioning method. The only information about the partitions that is needed by the querying system is the partition to which each record belongs. How these partitions are arrived at is irrelevant to the procedure we use to respond to a query.

The strategies of response to frequency, average and count queries based on the partitions are presented in Section 5.2. The formulas given in that section define the perturbed statistics as a function of the parameters governing the underlying partitions.

We have produced statistics from randomly generated queries. Perturbed values, true values and the relative errors were calculated. These random trials are described and the statistics which constitute the major results of the project are given in Section 5.3. Section 5.4 analyzes the results of these trials. Finally, the results of some attempts to compromise the database using trackers are presented in Section 5.5, and the implications of dynamically changing databases due to insertion and deletion of records

is discussed in Section 5.6.

5.2 Strategy of response

The basis of statistical security by means of partitioning is that statistics are only given for entire partitions rather than individual records. Exactly how we formulate such responses is shown in this section.

5.2.1 Average queries

The first type of query we will investigate is one which asks the average of values in a datafield; for all records with a characteristic formula C, written $AVG(C,j)$. The response to this kind of query is straightforward. Assume that for each partition we have calculated the average value of all the records in that partition for each datafield. This can be done as soon as the partitions are known. We then simply substitute the partition average for the true value of the record in answering the query. Formally stated, suppose all the records in the query set for a query Q belong to r different partitions; call these $G_1 \ldots G_r$. Let $C_1 \ldots C_r$ be the number of records belonging to the query set from each partition. We can define $B_i = \frac{C_i}{\sum_{\ell=1}^{r} C_\ell}$. Then our response to the query $Q = AVG(C,j)$ is $\sum_{i=1}^{r} B_i A_i$ where A_i is the average of all the records in partition G_i for the $j\underline{th}$ datafield.

For example, suppose there are eight records in the

query set, two of which belong to partition G_1, five to G_2 and one to G_3. Our response will be $\frac{2}{8}A_1 + \frac{5}{8}A_2 + \frac{1}{8}A_3$.

The amount of error introduced by this method depends on how much the true values in records differ from their partition averages as well as the proportion of records in a given partition which belong to the query set. The greater this proportion is, the closer to the true average will be the partition average. True answers are given by this strategy only when the query set consists of one or more complete partitions. This does not compromise the database since we are correctly responding only to whole partitions, not any subsets of partitions.

We can derive a rough upper bound for the difference between the true average and our response to an AVG query in terms of the maximum amount that any record pertaining to the query differs from its partitional average. Let us define that quantity to be $\epsilon = \max_i [\max_{V_q \in D_i} \{|A_i - V_q|\}]$, where i varies over the partitions involved in the query set $(1 \leq i \leq r)$ and V_q is a record belonging to the query set. The perturbed average is given by $\frac{1}{\sum_{i=1}^{r} C_i} \sum_{i=1}^{r} C_i A_i$ while the true average is $\frac{1}{\sum_{i=1}^{r} C_i} \sum_{i=1}^{r} \left(\sum_{q=1}^{C_i} V_q \right)$. The absolute difference between these quantities is, then,

$$\frac{1}{\sum_{i=1}^{r} C_i}\left|\sum_{i=1}^{r} C_i A_i - \sum_{i=1}^{r}\left(\sum_{q=1}^{C_i} V_q\right)\right| = \frac{1}{\sum_{i=1}^{r} C_i}\sum_{i=1}^{r}\left(\sum_{q=1}^{C_i} |A_i - V_q|\right).$$

This expression will be maximal if each absolute difference $|A_i - V_q|$ is equal to ϵ. The expression then becomes:

$$\frac{1}{\sum_{i=1}^{r} C_i}\sum_{i=1}^{r}\sum_{q=1}^{C_i}\epsilon = \frac{1}{\sum_{i=1}^{r} C_i}\sum_{i=1}^{r} C_i \cdot \epsilon = \epsilon .$$

Thus we conclude that the overall absolute error for an AVG query will be at most the maximum difference between any particular value and its partition average. For the rectangular partitioning method, ϵ will be less than the maximum range covered by a rectangle for the attribute queried.

5.2.2 Frequency queries

Queries which concern the size of the subgroup defined by a characteristic formula can be of two types. COUNT queries ask for the actual number of records in the subgroup whereas relative frequency queries ask only for the percentage of records pertaining to the query. This can be derived from the count simply by dividing by N. Frequencies are in general less compromising than counts because the user may not know the total number of records in the database. A frequency of .01 may mean one record if N = 100 or 10 records if N = 1000. It is for this reason that Denning [14] only gives answers to frequency queries in her random

sampling experiment. We answer count queries as well as frequencies and these will be discussed in the next section.

Let us assume, as in the previous section, that the records relevant to a query FREQ(C) belong to r different partitions. We will define n_i to be the actual number of records in the ith partition. Recall that C_i denotes the number of records in the query set which belong to the ith partition. Our response to the query FREQ(C) is

$$\text{Pfreq}(C) = \frac{C_1 + C_2 + \ldots C_r}{n_1 + n_2 + \ldots n_r} \cdot \frac{r}{s}$$ where s is the total number of partitions for the database.

The true response to this query can be written as

$$\text{tfreq}(C) = \frac{C_1 + C_2 + \ldots C_r}{n_1 + n_2 + \ldots n_r} \cdot \frac{n_1 + n_2 + \ldots n_r}{N}.$$ Therefore, the quantity pfreq(C) differs from the true response by a factor of $\frac{r}{s} \cdot \frac{N}{n_1 + n_2 + \ldots n_r} = P$. Let Z_q denote the average number of records per partition for the partitions involved in the query set of a query q, and let Z_N be the same average for the entire database. By definition $Z_q = \frac{n_1 + n_2 + \ldots n_r}{r}$ and $Z_N = \frac{N}{s}$. The perturbation factor P can be related to these quantities by rewriting the equation as $Z_N = P \cdot Z_q$. Clearly, if the average number of records per partition in the query set partitions is the same as for the entire database, P becomes one and pfreq(C) = tfreq(C). Otherwise P varies indirectly with Z_q. There-

fore if the variation in the size of the partitions is small then the statistics released for frequencies will be close to the true statistics. Furthermore, because not all the partitions will have exactly the same number of records, there will be some error introduced by this method, a fact which is essential to prevent statistical inference. It can also be seen that as r increases, that is, more partitions are involved in the query set, P becomes closer to one. This is due to the fact that as r approaches s, Z_q approaches Z_N. When r is small, the variations in partition size have a greater effect on the average Z_q. This is borne out by our results, as will be shown in the next section.

We now give an error analysis of the response to frequency queries. Let x be the absolute deviation in the average number of records per partition, i.e., $x = Z_q - Z_N$ or $x = \frac{n_1 + n_2 + \ldots n_r}{r} - \frac{N}{s}$. Pfreq(C) can then be written in terms of x: $\text{pfreq}(C) = \frac{C_1 + C_2 + \ldots C_r}{\left(\frac{N}{s} + x\right)s}$. We will define the function $f(x)$ to be the absolute error pfreq(C) − tfreq(C) related to the absolute deviation x.

So $$f(x) = \frac{C_1 + C_2 + \ldots C_r}{\left(\frac{N}{s} + x\right)s} - \frac{C_1 + C_2 + \ldots C_r}{N}$$

$$= (C_1 + C_2 + \ldots C_r)\left[\frac{1}{N + xs} - \frac{1}{N}\right]$$

$$f(x) = \frac{-(C_1 + C_2 + \ldots C_r)xs}{N(N + xs)}.$$

Let m denote the maximum number of records in any partition. The minimum number is of course the threshold, t. Then $t \leq \frac{N}{s} \leq m$, and the maximum deviation x will be either $x = t - \frac{N}{s}$ or $x = m - \frac{N}{s}$, since $f(x)$ is strictly decreasing. Evaluating $f(x)$ for these values we find:

$$f\left(t-\frac{N}{s}\right) = \frac{-(C_1+C_2+\ldots C_r)\,s\left(t-\frac{N}{s}\right)}{Nst} = \left(\frac{C_1+C_2+\ldots C_r}{N}\right)\left(\frac{\frac{N}{s}-t}{t}\right)$$

and

$$f\left(m-\frac{N}{s}\right) = \frac{-(C_1+C_2+\ldots C_r)\,s\left(m-\frac{N}{s}\right)}{Nsm} = \left(\frac{C_1+C_2+\ldots C_r}{N}\right)\left(\frac{m-\frac{N}{s}}{m}\right)$$

Since the relative error is equal to the absolute error divided by the true response, which is $\frac{C_1+C_2+\ldots C_r}{N}$ in this case, we find that the maximum absolute value of the relative error is either $\frac{\frac{N}{s}-t}{t}$ or $\frac{m-\frac{N}{s}}{m}$. It is not immediately obvious which of these quantities is larger, but it can be shown that if $m < \frac{Nt}{2st-N}$ then $\frac{\frac{N}{s}-t}{t}$ is the maximum relative error. Since $\frac{N}{s} \leq m$, $\frac{\frac{N}{s}-t}{t} \leq \frac{m-t}{t}$. On the other hand, if $m \geq \frac{Nt}{2st-N}$ then $\frac{m-\frac{N}{s}}{m}$ is the maximum relative error. Using a similar assumption, namely that $\frac{N}{s} \geq t$ we can show that $\frac{m-\frac{N}{s}}{m} \leq \frac{m-t}{m}$. Of these two

quantities, clearly $\frac{m-t}{t}$ is the larger. Thus we can say that the upper bound for the relative error of the perturbed response to frequency queries is $\frac{m-t}{t}$ or $\frac{m}{t} - 1$, where m is the maximum number of records in any query.

5.2.3 Count queries

A response to COUNT queries is simply the perturbed frequency multiplied by N, where N is the number of records in the database. In order to obtain an integer this quantity is then randomly rounded either up or down to the nearest integer. The formula for determining the response to a query count(C) can be formally stated as:

$$\text{pcount}(C) = \text{trunc}(\text{pfreq}(C) \cdot N + \text{roundbit}[\text{tcount}(C)]) .$$

The function trunc(x) truncates the decimal part of the real number x to give an integer result. One round bit is generated for each possible query set size (1 to N) and fixed before any querying takes place. The round bits are randomly generated ones or zeroes and the round bit used to determine the perturbed count is the one associated with the true count or query set size. This assures that the response to identical queries will always be the same, but different queries with the same query set size may have different responses if their values for the perturbed frequency are different. Similarly, two queries with the same perturbed frequency may have different responses to count queries if their query set sizes differ. This makes it hard to determine

which way a count has been rounded by examining the frequency response.

Since the response to count queries is derived from the perturbed frequency, it will have similar properties in terms of accuracy and maximum relative error.

5.3 Experimental results

We have tested partitioned databases of size $N = 100$, $N = 500$ and $N = 1000$ for statistical accuracy. Each record consisted of four attribute values and four data fields, and were created using a pseudo-random number generator. The data fields were related to their corresponding attribute value in that they lie within an interval defined by that value. For instance, if the value for attribute one of the ith record is 3 then the value for the first data field of that record is in the range [201,300].

Three hundred randomly generated characteristic formulas were used to measure the error in the statistics for each database. Two values of t were used: 3 and 5. The relative error of the partition-based statistics to the true statistics were calculated for each query. All the queries were classified into 10 intervals by query set size and mean relative errors were calculated for each interval. These results are summarised later in this section.

5.3.1 Generating random queries

A characteristic formula consists of a number of specific values for each attribute related by the logical operators <u>and</u>, <u>or</u> and <u>not</u>. If no value is specified for a

given attribute then we don't care about that attribute; the other attributes will specify which records belong to the query set.

The first step in generating a random characteristic formula is to choose the number of values to be specified for each attribute. This ranges from 0 (don't care) to $d_i - 1$ for the ith attribute, where d_i is the number of possible values for that attribute. Once we know the number of values to be chosen for an attribute, we then pick that number of different values. If more than one value is chosen, we will interpret this to mean any of those values can match the value of a record for the same attribute. This simply means that we can insert the operator <u>OR</u> between the different values. Suppose, for example, we have chosen the values 1, 3 and 4 for some attribute i. Then any record which has value 1 or 3 or 4 will match this query at the ith attribute. Clearly, since each record has only one value per attribute, there is no other way of interpreting this.

The final step in our characteristic formula generation is to choose the operators which go before each attribute's set of values. We needn't concern ourselves with the operator <u>NOT</u> since any negated set of values is equivalent to its complement - all the other values for that attribute - and any set of values is equally likely. Any attribute which has no values specified does not require an operator since we don't care about it. For all the other attributes we need simply to choose the operator <u>AND</u> or <u>OR</u>.

The interpretation of these operations is not as simple.

There are five general cases for four attributes, corresponding to the number of AND operators in the formula. Let us consider each case in turn. We will refer to the set of values chosen for the ith attribute as S_i. If there are no AND's the interpretation is straightforward:
$(A_1 = S_1)$ OR $(A_2 = S_2)$ OR $(A_3 = S_3)$ OR $(A_4 = S_4)$.
Suppose the first attribute only is associated with an AND. Then we have: $(A_1 = S_1)$ AND $[(A_2 = S_2)$ OR $(A_3 = S_3)$ OR $(A_4 = S_4)]$. A similar formula will occur if one of the other attributes is the only one associated with an AND.

The interpretation of a formula containing two AND operators is not as clear. We have chosen the following: $(A_1 = S_1)$ AND $(A_2 = S_2)$ AND $[(A_3 = S_3)$ OR $(A_4 = S_4)]$. This follows logically from the previous formula and is more restrictive, i.e., there will be fewer records satsifying this formula than the previous case, given the same S_i's.

Next we consider the meaning of a formula with three AND's. We will take this to mean the following:
$[(A_1 = S_1)$ AND $(A_2 = S_2)$ AND $(A_3 = S_3)]$ OR $(A_4 = S_4)$.
Finally, if all the operators are AND we have, of course, the formula $(A_1 = S_1)$ AND $(A_2 = S_2)$ AND $(A_3 = S_3)$ AND $(A_4 = S_4)$.

The order of the attributes in any of the formulas can be changed from the examples given above depending on which attributes happen to be associated with an AND operator. Figure 5.1 summarises the interpretations in a more general manner.

p , q , r , and s each stand for the expression $(A_i = S_i)$, where each i is a different integer in the range $1 \leq i \leq k$.

number of AND's	formula
0	p OR q OR r OR s
1	p AND [q OR r OR s]
2	p AND q AND [r or s]
3	p OR [q AND r AND s]
4	p AND q AND r AND s

Figure 5.1 Interpretations of Characteristic Formulas with AND and OR Operators for $k = 4$.

If any attribute i has no particular values chosen for it, we can simply set S_i to all the values for that attribute. The meaning of the formulas will remain unchanged for all the situations except one. If there are three AND operators and each one is associated with a "don't care" then the fourth attribute is meaningless and every record will belong to the query set. We have, in effect, the following situation: (Don't care) or $(A_i = S_i)$. This formula is a tautology and to avoid this we interpret this special case to be simply $(A_i = S_i)$.

Clearly, not all logical formulae using AND and OR operators are possible using this query language. To admit any conceivable query would require a much more complex system, however, and it is not the purpose of this thesis to design a comprehensive query language and interpreter. Our objective was to be able to generate queries at random such that the size of the query sets would be distributed as uniformly as possible over the interval $[0,N]$, and this has been accomplished.

5.3.2 Statistics produced

Each randomly generated characteristic formula C was posed as the queries FREQ(C), AVG(C,j) for all datafields j, and COUNT(C). Both the true values and the perturbed values using the formulas given in Section 5.2 were calculated for the queries.

The relative error of the perturbed values based on the partitions was determined for all frequency and average

queries. Let tval denote the true answer to a query and pval the perturbed response. The relative error is given by $\frac{pval - tval}{tval}$.

When all the queries were posed for a particular database, they were classified into 10 equal intervals of [0,N] according to the number of records pertaining to the query. For each interval, the mean of the absolute value of the relative errors was calculated for both query types.

5.3.3 Results

In the following tables the four digits in brackets are the sizes of the four domains. Rectangular partitioning is project 1 while project 2 is the hierarchical method.

Results for N = 100 T = 3 (5342)

Project 1

Query set size	Number of queries	Pfreq rel. error	Pavg rel. error
1-10	24	.083	.134
11-20	21	.052	.077
21-30	28	.041	.060
31-40	33	.025	.057
41-50	39	.024	.044
51-60	36	.021	.035
61-70	27	.016	.018
71-80	44	.009	.015
81-90	31	.007	.011
91-100	17	.002	.004
Total	300	.026	.043

total partitioning time: 346 msec

number of partitions: 30 (91%)

Project 2

Query set size	Number of queries	Pfreq rel. error	Pavg rel. error
1-10	24	.094	.117
11-20	21	.056	.081
21-30	28	.038	.055
31-40	33	.037	.057
41-50	39	.025	.041
51-60	36	.020	.033
61-70	27	.014	.018
71-80	44	.014	.013
81-90	31	.009	.011
91-100	17	.002	.005
		.029	.040

total partitioning time: 163 msec

number of partitions: 27 (82%)

Results for N = 100 T = 5 (5342)

Project 1

Query set size	Number of queries	Pfreq rel. error	Pavg rel. error
1-10	24	.097	.288
11-20	21	.041	.206
21-30	28	.033	.147
31-40	33	.030	.119
41-50	39	.020	.074
51-60	36	.016	.069
61-70	27	.015	.045
71-80	44	.005	.036
81-90	31	.004	.024
91-100	17	.000	.008
		.024	.094

total partitioning time: 282 msec
number of partitions: 17 (85%)

Project 2

Query set size	Number of queries	Pfreq rel. error	Pavg rel. error
1-10	24	.064	.222
11-20	21	.045	.187
21-30	28	.032	.092
31-40	33	.024	.086
41-50	39	.015	.092
51-60	36	.016	.052
61-70	27	.014	.022
71-80	44	.008	.025
81-90	31	.005	.018
91-100	17	.001	.008
		.020	.075

total partitioning time: 86 msec
number of partitions: 14 (70%)

Results for N = 100 T = 3 (5425)

Project 1

Query set size	Number of queries	Pfreq rel. error	Pavg rel. error
1-10	25	.105	.273
11-20	33	.056	.096
21-30	27	.028	.098
31-40	34	.030	.053
41-50	30	.022	.041
51-60	32	.016	.038
61-70	30	.014	.029
71-80	26	.009	.020
81-90	37	.005	.015
91-100	26	.002	.007
		.028	.067

total partitioning time: 497 msec
number of partitions: 29 (88%)

Project 2

Query set size	Number of queries	Pfreq rel. error	Pavg rel. error
1-10	25	.118	.212
11-20	33	.074	.084
21-30	27	.052	.090
31-40	34	.042	.041
41-50	30	.026	.035
51-60	32	.022	.039
61-70	30	.020	.025
71-80	26	.014	.021
81-90	37	.007	.014
91-100	26	.001	.006

total partitioning time: 151
number of partitions: 27 (82%)

Results for N = 100 T = 5 (5425)

Project 1

Query set size	Number of queries	Pfreq rel. error	Pavg rel. error
1-10	25	.098	.269
11-20	33	.069	.135
21-30	27	.048	.151
31-40	34	.045	.077
41-50	30	.030	.067
51-60	32	.026	.058
61-70	30	.015	.043
71-80	26	.009	.032
81-90	37	.002	.021
91-100	26	.000	.009

total partitioning time: 360 msec
number of partitions: 16 (80%)

Project 2

Query set size	Number of queries	Pfreq rel. error	Pavg rel. error
1-10	25	.110	.185
11-20	33	.066	.108
21-30	27	.052	.127
31-40	34	.036	.045
41-50	30	.019	.033
51-60	32	.021	.031
61-70	30	.015	.029
71-80	26	.011	.024
81-90	37	.003	.019
91-100	26	.001	.006

total partitioning time: 188
number of partitions: 16 (80%)

Results for N = 500 T = 3 (6295)

Project 1

Query set size	Number of queries	Pfreq rel. error	Pavg rel. error
1-50	28	.059	.110
51-100	35	.032	.029
101-150	28	.023	.018
151-200	29	.015	.026
201-250	33	.011	.017
251-300	33	.010	.016
301-350	23	.007	.010
351-400	34	.005	.007
401-450	25	.003	.004
451-500	32	.001	.002

total partitioning time: 8087 msec
number of partitions: 137 (82.5%)

Project 2

Query set size	Number of queries	Pfreq rel. error	Pavg rel. error
1-50	28	.049	.059
51-100	35	.040	.033
101-150	28	.034	.022
151-200	29	.025	.030
201-250	33	.013	.016
251-300	33	.016	.014
301-350	23	.010	.006
351-400	34	.009	.006
401-450	25	.007	.003
451-500	32	.003	.002

total partitioning time: 958 msec
number of partitions: 124 (75%)

Results for N = 500 T = 5 (6295)

Project 1

Query set size	Number of queries	Pfreq rel. error	Pavg rel. error
1-50	28	.053	.237
51-100	35	.035	.072
101-150	28	.022	.041
151-200	29	.019	.068
201-250	33	.016	.039
251-300	33	.012	.037
301-350	23	.008	.022
351-400	34	.006	.015
401-450	25	.004	.008
451-500	32	.001	.004

total partitioning time: 5,743 msec

number of partitions: 81 (81%)

Project 2

Query set size	Number of queries	Pfreq rel. error	Pavg rel. error
1-50	28	.066	.137
51-100	35	.042	.072
101-150	28	.025	.037
151-200	29	.021	.073
201-250	33	.017	.041
251-300	33	.009	.034
301-350	23	.014	.012
351-400	34	.008	.011
401-450	25	.008	.006
451-500	32	.001	.004

total partitioning time: 571 msec

number of partitions: 70 (70%)

Results for N = 500 T = 3 (9459)

Project 1

Query set size	Number of queries	Pfreq rel. error	Pavg rel. error
1-50	42	.034	.071
51-100	41	.023	.060
101-150	31	.017	.038
151-200	31	.016	.029
201-250	23	.012	.028
251-300	31	.010	.020
301-350	22	.006	.015
351-400	26	.005	.010
401-450	31	.003	.007
451-500	21	.001	.003

total partitioning time: 11.718 sec

number of partitions: 143 (86%)

Project 2

Query set size	Number of queries	Pfreq rel. error	Pavg rel. error
1-50	42	.075	.117
51-100	41	.044	.059
101-150	31	.029	.041
151-200	31	.023	.036
201-250	23	.022	.024
251-300	31	.014	.026
301-350	22	.012	.017
351-400	26	.007	.013
401-450	31	.004	.009
451-500	21	.001	.003

total partitioning time: 1.167 sec

number of partitions: 125 (75%)

Results for N = 500 T = 5 (9459)

Project 1

Query set size	Number of queries	Pfreq rel. error	Pavg rel. error
1-50	42	.034	.114
51-100	41	.022	.091
101-150	31	.018	.068
151-200	31	.014	.050
201-250	23	.010	.044
251-300	31	.006	.035
301-350	22	.004	.022
351-400	26	.003	.018
401-450	31	.001	.012
451-500	21	.001	.004

total partitioning time: 10,303 msec

number of partitions: 82 (82%)

Project 2

Query set size	Number of queries	Pfreq rel. error	Pavg rel. error
1-50	42	.058	.141
51-100	41	.023	.079
101-150	31	.018	.051
151-200	31	.015	.044
201-250	23	.012	.034
251-300	31	.011	.030
301-350	22	.008	.022
351-400	26	.007	.015
401-450	31	.003	.011
451-500	21	.001	.004

total partitioning time: 861 msec

number of partitions: 72 (72%)

Results for N = 1000 T = 3 (9554)

Project 1

Query set size	Number of queries	Pfreq rel. error	Pavg rel. error
1-100	51	.053	.056
101-200	37	.022	.033
201-300	26	.017	.023
301-400	25	.013	.023
401-500	27	.009	.017
501-600	20	.009	.013
601-700	23	.008	.011
701-800	28	.004	.006
801-900	33	.003	.007
901-1000	30	.001	.002
		.017	.022

total partitioning time: 42,176 msec

number of partitions: 278 (83%)

Project 2

Query set size	Number of queries	Pfreq rel. error	Pavg rel. error
1-100	51	.067	.068
101-200	37	.046	.037
201-300	26	.029	.018
301-400	25	.029	.036
401-500	27	.019	.017
501-600	20	.018	.014
601-700	23	.012	.011
701-800	28	.008	.008
801-900	33	.006	.006
901-1000	30	.003	.002
		.027	.025

total partitioning time: 1669 msec

number of partitions: 224 (67%)

Results for N = 1000 T = 5 (9554)

Project 1

Query set size	Number of queries	Pfreq rel. error	Pavg rel. error
1-100	51	.047	.125
101-200	37	.021	.055
201-300	26	.023	.055
301-400	25	.010	.044
401-500	27	.009	.029
501-600	20	.007	.022
601-700	23	.006	.020
701-800	28	.005	.011
801-900	33	.002	.010
901-1000	30	.001	.003
		.016	.044

total partitioning time: 18,609 msec

number of partitions: 171 (85%)

Project 2

Query set size	Number of queries	Pfreq rel. error	Pavg rel. error
1-100	51	.080	.099
101-200	37	.033	.057
201-300	26	.027	.038
301-400	25	.024	.064
401-500	27	.017	.027
501-600	20	.012	.020
601-700	23	.008	.015
701-800	28	.006	.012
801-900	33	.004	.008
901-1000	30	.001	.003
		.026	.039

total partitioning time: 1715 msec

number of partitions: 143 (71%)

5.4 Analysis of the results

The statistics produced using partitioned databases can be analyzed in several ways. First, we will compare the two different partitioning methods, not only in terms of the actual statistics in response to the same set of queries but also the computing time taken to produce the partitions. We will also examine the statistics in relation to N and the threshold, t. Finally, we will consider a variation of the hierarchical method which takes the probability of querying a given attribute into account.

5.4.1 Rectangular versus hierarchical partitioning

It is readily apparent that the hierarchical partitioning method produces fewer partitions than the rectangular bottom-up method. Whereas the number of partitions formed by rectangular partitioning in project 1 is consistently greater than 80% of the maximum number possible, the second project produces roughly 70%. This would lead us to expect worse results from the second project, especially the frequency statistics.

As the number of partitions decreases, the variability in their size increases and this will lead to greater errors in frequency response. This is in fact the case as frequency errors are consistently slightly higher for the hierarchical method. The difference is not great, however, and with one exception all the frequency errors are less than 10%, and decrease rapidly with increasing query set size.

We may also expect the AVG query response to be worse

for the second project simply because of the greater number of records per partition, leading to a greater variation in the data values within a given partition. The response to AVG queries, however, is also dependent on the homogeneity of the partitions. Even if there are more records in a partition on the average, if those records are similar to each other the variation in the data values may be less. We have hypothesized at the end of Chapter 4 that the statistics would be as good for the splitting algorithm as for the rectangular merging even though fewer partitions are formed. The results uphold this hypothesis. The errors for AVG queries are at least as low for the second project as compared to the first if not lower. Again, the difference is not great, but the fact that partitionings can produce better statistics with fewer numbers of groups means that these groups are well formed. This addresses a major criticism of partitioning as a method to protect statistical databases, namely that groups can be ill-formed and the statistics produced thereby are not meaningful. Our hierarchical partitioning method does produce meaningful statistics with errors generally less than 10%.

A final means of comparison between the two projects is partitioning time. We have stated that the rectangular partitioning method has a thoeretical worst case complexity of $O(kN^2 \log N)$ whereas the hierarchical method has complexity of $O(kN)$. The results bear these figures out and show the rather large difference between the two as N increases

(see Figure 5.2). The second project is initially faster than the first for N of 100, by at least a factor of two. As N increases by a factor of five, the partitioning time for project two increases by nearly the same amount, indicating that it is indeed linear. The rectangular partitioning time, on the other hand, increases by more than 20 times. When N is 10 times the initial value, the first project is over 100 times slower while the second project is almost exactly 10 times slower. Thus the first project is running in time proportional to N^2. This is a significant difference as is evident by the fact that for identical databases with N = 1000, project 1 requires over 40 seconds more CPU time than project 2. The overwhelming discrepancy in partitioning times in favor of hierarchical partitioning supercedes any other criterion in choosing the better method.

Not only is the hierarchical method much faster, but the partitions are more homogeneous and the AVG query errors are slightly better. Only the FREQ query errors are worse than those using rectangular partitioning and these are certainly within a tolerable range.

5.4.2 Effect on the statistics of the number of records

One would expect the statistics to improve as N increases because the size of the partitions stays the same. Therefore the number of records in a partition is a smaller proportion of N. As the total number of partitions increases, the variability in their sizes should decrease slightly and improve the frequency statistics. Figure 5.3

	time in msec			relative increase from N=100	
[redacted]	Project 1	Project 2	N	Project 1	Project 2
100	374	163			
500	8084	958	5	21.62	5.87
1000	42176	1669	10	112.77	10.23

Figure 5.2 Partitioning CPU Times

N	t	Project #	Pfreq rel. error	Pavg rel. error
100	3	1	.026	.043
100	3	2	.029	.040
1000	3	1	.017	.022
1000	3	2	.027	.025
100	5	1	.024	.094
100	5	2	.020	.075
1000	5	1	.017	.044
1000	5	2	.026	.039

Figure 5.3 Selected mean errors for all queries

gives the average errors for all queries for some selected databases. These figures give an indication of the overall accuracy of the statistics. They show that response to AVG queries does improve significantly with increased N and FREQ queries improve less dramatically.

5.4.3 Choice of the partitioning threshold

Another important aspect of partitioning is the choice of a threshold or minimum number of records per partition. We want to pick a threshold which is large enough to ensure security yet one which does not cause overly inaccurate statistics to be produced. We have tested identical databases with identical queries for both projects using thresholds of $t = 3$ and $t = 5$. The results for $t = 3$ should be better in all cases because of the greater number of partitions for frequency queries and the fewer number of records per partition giving better data averages for AVG queries.

The results show that response to AVG queries is much better for $t = 3$ as opposed to $t = 5$. In fact, for small N the larger threshold produced statistics which were not at all accurate. This is entirely in keeping with our theory. We would also expect the difference to be less marked for larger N, and such is the case.

The frequency response showed something quite unexpected. Differences between the results for the two thresholds were much less than we expected. In fact, in most cases they were minimal. This is surprising at first

because one would expect fewer partitions to have a greater variation in size. While this may be so, it need not be much different, but we must recall the theoretical maximum error for frequencies given in Section 5.2.2. The formula is $\frac{m}{t} - 1$ where m is the maximum number of records in any partition. Thus while m will certainly increase when t increases, it will not necessarily increase by a greater proportion than t. This explains the initially puzzling decreased accuracy in frequency statistics with decreased t.

The other criterion in choosing a threshold is the degree of security provided. This will be discussed in more detail in the next section. We have already stated the theoretical security inherent in a partitioning system, and the results of tracker attacks confirm this. They also show, however, that there is a difference in the number of successful attacks for different values of t.

Choosing a threshold is a decision which must be carefully considered. There is a definite tradeoff between accuracy of statistics (for averages) and compromisability. For small N, the statistics produced with the larger threshold are not very accurate and this indicates that a smaller threshold be used. This difference is not as marked for larger N, but the statistics are nevertheless worse for the larger t. There is less threat of compromisability with larger t but that threat is not great in either case.

5.4.4 Hierarchical partitioning using probabilistic queries

We have tested the hierarchical splitting method using probabilistic queries as explained in Section 4.2.2 There are two methods of ordering the attributes to be used for splitting. One is to use the probability that an attribute be specified in a query as the sole criterion ("prob" variation) and the other is to separate the attributes into more probable and less probable ones and then order them by decreasing size ("mixed" variation).

The probabilities were randomly assigned to each attribute. These were then used to order the attributes. Let us call p_i the probability for attribute i $(0 \leq p_i \leq 1)$. Random characteristic formulas were than generated according to the probabilities. One or more values were specified for an attribute i only if a random number was greater than p_i. Otherwise a "don't care" was generated. This was done for all k attributes. We can denote the overall query probability for query q_j as $qp_j = \Pi(1 - p_i)\Pi p_r$ where i ranges over all attributes not specified and r ranges over all attributes specified.

Three hundred such characteristic formulae were then posed and the statistics generated. When calculating mean relative errors weighted sums were used. That is, if f_i is the frequency error for query q_i, the mean relative error for all queries q_j whose query set size belongs in the same interval is given by $\frac{\Sigma qp_j \cdot f_j}{\Sigma qp_j}$. A similar formula

was used for AVG queries as well. Table 5.4 gives the results of one such trial for both the "prob" and "mixed" variations with $N = 100$ and $t = 3$.

The results of these trials show that the frequency errors seem to be a bit worse than the original method, but the AVG errors are much better in both cases.

Again we find the situation where the number of partitions and variability within the partitions govern the accuracy of the frequency statistics.

These particular variations reorder the attributes from the original method which was designed to create the most partitions by ordering the attributes by size. We would then expect the "mixed" variation to be better in this respect and indeed it is, but still worse than the original.

The main purpose of these variations is to decrease the error for AVG queries. Both variations succeed handily at this. In fact, they both produce the best statistics for AVG queries of any trial at any level of N or t. There is little difference between the two.

We can conclude that, if the probabilities for specifying attributes are known beforehand, using the "prob" or "mixed" variations of hierarchical partitioning will give highly accurate statistics. The best method overall is the "mixed" variation because it produces more partitions and the frequency errors are lower than the variation which uses only attribute probabilities to order the attributes.

Table 5.4 Results for Project 2 using Probabilistic Queries

N = 100 T = 3 (5342)

"Prob" variation

Query set size	Number of queries	Pfreq rel. error	Pavg rel. error
1-10	4	.103	.065
11-20	12	.156	.016
21-30	5	.039	.008
31-40	34	.080	.001
41-50	30	.041	.004
51-60	25	.041	.004
61-70	81	.039	.001
71-80	21	.039	.004
81-90	42	.015	.001
91-100	45	.001	.000
		.040	.003

total partitioning time: 165 msec

number of partitions: 26 (79%)

"Mixed" variation

Query set size	Number of queries	Pfreq rel. error	Pavg rel. error
1-10	4	.119	.012
11-20	12	.078	.044
21-30	5	.068	.014
31-40	34	.055	.002
41-50	30	.029	.005
51-60	25	.028	.036
61-70	81	.023	.004
71-80	21	.018	.004
81-90	42	.013	.004
91-100	45	.001	.000
		.026	.008

total partitioning time: 182 msec

number of partitions: 27 (82%)

5.5 Compromise

Partitioned databases control compromise by preventing a questionner from isolating the information contained in an individual record. Whether it is queried directly or inferred from linear combinations of queries, the individual's sensitive data cannot accurately be determined. The reason why no compromise is theoretically possible is that we are generating the statistics on the basis of the partitions instead of the records themselves.

When calculating the average of a set of records we use each relevant partition's average instead of the true values. No amount of querying, no matter how clever, will be able to determine the true value since it was never used in the first place. Frequencies are also derived using a formula based on the partitions. This response can be accurate if the number of relevant records per partition is the same as the total number of records divided by the total number of partitions. Since the user is unaware of the partitions or at least of which records belong to which partitions, he has no way of knowing an accurate answer from a perturbed one. Furthermore, since the total number of records in the database is never released, a user cannot translate relative frequencies into actual counts. If we do permit count queries it may be possible to determine N, but the counts are derived from the frequencies and randomly rounded up or down. It is still impossible, therefore, to tell an accurate response from an inaccurate one.

In order to test the security of our calculations which in theory will not permit compromise, we have attacked the database with randomly generated general trackers. The trackers attempt to isolate a single record and determine its data values.

5.5.1 Random tracker generation

According to Denning, Denning and Schwartz [11] a general tracker is any characteristic formula T whose query set size is in the range [2k,N-2k] where k is the size of the smallest allowable query set. Although we have no restriction on the query set size as such, we will not release a count if it is less than the threshold t. We may then consider a general tracker to be any characteristic formula whose query set size is in the range [2t,N-2t]. It is not hard to randomly generate such a formula. The method we have chosen is given by the following algorithm:

ALGORITHM TRACKGEN

1. Choose any two attributes A_i and A_j.
2. Select a value at random for both A_i and A_j; call these V_i and V_j respectively.
3. Formulate the characteristic formula T as $(A_i = V_i)$ OR $(A_j = V_j)$, for $i \neq j$.

The formula T is guaranteed to have a query set size in the desired range, and by choosing two different values we have avoided generating the same formula too many times.

It remains to find a target. The target record must be a single isolatable record. That means, it must be uniquely

identifiable by its characteristic formula, C . To find such a record, we simply generate formulae of the type:

$$(A_1 = V_1) \text{ AND } (A_2 = V_2) \text{ AND } (A_3 = V_3) \text{ AND } (A_4 = V_4) ,$$

where each V_i is a single value in the range $[1,d_i]$ chosen at random. We then determine the number of records which match this characteristic formula. If there is only one, that is our target. If not, we repeat the procedure until we do find an isolable record.

The tracker formula T and the target formula C are not, and should not be, related. It may happen that the same tracker is generated for different targets or different trackers are generated for the same target.

Once the tracker and target have been determined, we pose the queries which are intended to infer the existence and the particular data values of the target record. These queries come from the tracker equation given in Chapter 2, namely

$$q(C) = q(C + T) + q(C + \overline{T}) - q(T) - q(\overline{T}) .$$

Inferred values for the frequency, count, and averages are calculated using the perturbed responses based on the partitions. These are compared with the actual values of the target record and relative errors determined.

5.5.2 Results of tracker attacks

The results of attempts to compromise the database using trackers are given below. We have counted the number of times the inferred frequency is within 10% of the actual frequency. For those attacks which have inferred the

Table 5.5 Tracker Results

	no. of attacks	freq. within 10%	avg. within 10%	count = 1
project 1	100	6	5	19
project 2	100	3	3	18
a) Results for N = 100, T = 3				
project 1	100	2	1	13
project 2	100	2	2	23
b) Results for N = 100, T = 5				
project 1	50	2	1	4
project 2	50	2	3	2
c) Results for N = 500, T = 3				
project 1	50	0	0	4
project 2	50	2	3	7
d) Results for N = 500, T = 5				
project 1	50	2	1	4
project 2	50	1	1	1
e) Results for N = 1000, T = 3				
project 1	50	0	0	1
project 2	50	0	0	2
f) Results for N = 1000, T = 5				

frequency accurately we count the number of datafield values, as many as four per record, which are also within 10% of the true value. If a datafield has been accurately inferred but not the frequency, then the user has learned nothing about the target record because he cannot tell for certain that there is only one record with that particular characteristic formula. Indeed, as we mentioned previously, it may be hard for the user to determine the frequency which is equivalent to a count of exactly one. If this is the case then trackers will be of little use to him anyway. We have also included count queries using random rounding and have tallied the number of tracker attacks which infer a correct count of one.

There were no attacks which determined the frequency and all the datafields to within 10% although several trackers inferred three of the four datafileds. Many times the inferred values were negative and the relative errors reached as much as 1300%.

5.5.3 Analysis of tracker attacks

The tracker attacks inferred very few correct frequency values; less than 5% of these were accurately determined for nearly all the series of attacks. Even fewer data values were inferred. Since there are four such values for each record, the actual percentage of data values accurately inferred was less than 1% in almost all cases.

With such a low number of successful inferences using trackers, they become virtually useless. It is impossible to tell which records and which trackers will yield accurate

information. A tracker which infers the frequency and several data values for a record in one project does not accurately infer any information about the record in a different project or for a different value of t. Whether or not a particular tracker compromises a particular record seems to be largely due to chance.

A more disturbing result is evident for counts, especially with a low number of records. Almost 20% of the inferred counts equalled one for N = 100 records. While the results were much better for larger N, they were always higher than the number of compromised frequencies. We must conclude that random rounding, even when used with the perturbed frequencies, does not provide an adequate means of protection. This is similar to a result given anecdotally by Denning [14], who studied tracker attacks using a "simple rounding control". Although Denning fails to give any figures to substantiate it, Denning states that "... the tracker attacks were more likely to subvert the rounding control than the random sample control. In many cases, the tracker revealed the exact value despite the rounding control."[4]

There appears to be little difference between the rectangular and hierarchical partitioning methods in terms of

---

[4]D.E. Denning, "Secure Statistical Databases with Random Sample Queries", ACM TODS, 5 (1980), p.307.

the number of values inferred.

We would expect the results to be better for $t = 5$ as opposed to $t = 3$. This is indeed the case as shown by the results. For $N = 100$ there were 9 frequencies inferred for both projects at $t = 3$ while only 4 were inferred for $t = 5$. The difference is not as great for $N = 500$ (4 versus 3) but were even more dramatic for $N = 1000$ where not one frequency value was inferred for either project with $t = 5$.

The results also show a slight improvement with increasing numbers of records. While 13 frequencies and 11 data values were inferred for $N = 100$, only 7 frequencies and 6 data values were determined for $N = 500$, and 3 frequencies and 2 data values were compromised for $N = 1000$.

In summary, statistics produced from a partitioned database are theoretically secure in that they are based on the partitions rather than the individual records. Our random tracker attacks show that some trackers do partially compromise some records, but at a low rate, and it is impossible to predict which trackers or records will be involved in that compromise. Many such attacks yield results which are off by several hundred percent and are completely meaningless. In fact, the average error for inferred frequencies was 1.35 for $N = 100$, 3.72 for $N = 500$ and 6.41 for $N = 1000$ while for data values the average errors were 9.95 for $N = 100$, 2.57 for $N = 500$ and 1.74 for $N = 1000$. Thus we conclude that partitioned databases are secure from tracker

attacks using frequency and average queries.

5.6 Changes in the database

Some databases being kept for statistical purposes are static databases. Once formed, insertions, deletions or updates are never performed on any records. The census database is a prime example of a static database. Other statistical databases such as medical or personnel databases may require periodic updating of the records. This can involve changes to data values or attributes within a record, inserting entirely new records or deleting records from the database.

In this section we outline the procedures which can be followed to implement changes in a partitioned database. It is desirable to avoid reforming the partitions if possible, but we must maintain the security and integrity of the database above all other considerations.

5.6.1 Insertions

Inserting a new record into the database involves determining which existing partition should contain the record. This is easily done with hierarchical partitions. All we need do is follow the appropriate path to a leaf node according to the attribute values of the new record. The method is simple and requires at most k comparisons.

Rectangular partitions pose more problems for inserting new records. Initially we must start with the first iteration of the procedure involving the first two attributes. In order to find the rectangle in which the new record belongs, we have

to inspect each rectangle until we find the one which covers the cell corresponding to the record's values. If no rectangle covers that cell we must insert the record into the nearest rectangle. This procedure is repeated for each iteration, requiring $O(ks)$ comparisons where s is the number of rectangles formed and is proportional to N. If the ranges covered by the final rectangles for each attribute are kept we need only inspect those rectangles, but k comparisons are still needed for each rectangle. There we see another advantage of the hierarchical method.

Once the correct partition is found and the new record inserted, we need only update the size of the partition and its data value averages. Queries are then answered just as before. We may designate a maximum allowable partition size such that if any partition grows larger than that size, we will repartition the entire database. This will prevent the statistics from becoming too inaccurate.

5.6.2 Deletions

Deleting a record can be accomplished simply by eliminating it from the partition to which it belongs and updating the size and data value averages for that partition. This is just as easily accomplished in either partitioning method.

The problem arises when the deletion of a record causes the size of a partition to fall below t.

In order to avoid repartitioning the database each time this happens (and possibly allow for an insertion to

rectify the situation) we propose the following scheme for producing statistics with undersized partitions. Consider the two cases of queries involving such a partition, which we will designate as P*. In the first case, the query set is entirely from P*, while in the second case the query set involves records from other partitions as well. For case 1 we answer queries as if the deleted record were still included in P*. We consider the size to be t and use the original data value averages. It is clear that this is necessary to maintain the security of the database.

If a query falls into the second category we can use updated data averages for P* based on the remaining records, but we will still consider the size of the partition to be the minimum, t. This method does require some overhead of maintaining two sets of data averages and discerning the two different cases. The statistics produced will be more current, however. If the number of partitions which have fallen below the minimum size becomes greater than some predefined threshold we will repartition the database.

5.6.3 Updates

If a record needs to be updated, it is changed without affecting the configuration of the partitions or the response strategy. Only the data averages of that record's partition may have to be recalculated.

It should be noted that if any of these changes to records, insertions, deletions or updates, is made to just one record it may jeopardize the security of the database. This

is simply because the same query may give different results before and after the change. Under our system of response it should be very hard to determine which record has been altered and by how much, since data averages have been changed and it could be any member of the partition which caused the change. Nevertheless it is generally safer to perform a number of changes to records at the same time. This may also save time as several changes may occur to records in the same partition and the resulting updates need only be performed once.

CHAPTER SIX

CONCLUSIONS

6.1 Effectiveness of partitioning to control inference

We have shown partitioning a database into small disjoint groups of records to be an effective method for securing against statistical inference. The statistics generated using partitions are accurate and there is no possibility of isolating any individual record. Partitioning can provide meaningful statistics at a low cost incurred primarily when setting up the database.

The two partitioning schemes presented in this thesis divide the database into small groups, which helps to provide accurate statistics. When statistics are calculated using the groups instead of individual records, those groups should be small and the records within a group as similar as possible. One major criticism of this technique is the questionable ability to provide meaningful statistics [9,10, 11,15]. We have proven that there is a simple method for producing small homogeneous groups by successively splitting the database on the basis of the different values of the domains. The results of our query trials show that the statistics given out are comparable to those produced by random sample queries even though the data fields were related to the attribute values, whereas in [15] they were random values uniformly distributed over a fixed interval.

Security against personal disclosure is guaranteed

in our system. It is impossible to isolate a record and a user can at best guess the value of any individual information with no way of telling when he is correct. Results of queries involving few records can be given at no risk, although they are prone to be more inaccurate than results from large query sets. This is, in fact, desirable from the standpoint of security since these queries are the most revealing of individual data.

Another advantage of partitioning is that there is no additional cost in answering queries once the partitioning has been performed and group averages calculated. The appropriate group average is merely substituted for the actual value of the record when answering a query. This contrasts greatly with the random sampling technique used by Denning [15] which calculates the sample upon answering a query for every query, although it is a relatively fast procedure. Hierarchical partitioning provides a very fast method for splitting a database in linear time proportional to the input size and number of attributes.

We have proposed a method for using partitions in a dynamic environment, another major criticism of this technique. It cannot always provide for instant integration of new data, especially when partitions fall below the minimum number of records required. Changes which can be introduced without affecting the partitions are done immediately. Under such a system periodic repartitioning may have to be done, but this should not be a major cost and could be performed as part of a

periodic updating of the database.

6.2 Contributions of this thesis

This project is the first implementation of a partitioning scheme for use in statistical databases. We have proven that partitioning is a viable method for securing against statistical inference, and can give meaningful statistics to legitimate users. We have designed a fast, simple and effective algorithm for generating the partitions by means of hierarchical splitting.

We have also implemented a modified version of Yu and Chin's algorithm [33] which forms disjoint rectangular regions which cover a two-dimensional matrix. This algorithm provides for a high percentage of the total number of partitions possible for a given database. Unfortunately it is rather costly, running in time proportional to the square of the input data size and having a worst case time complexity of $O(N^3)$. It is not necessary to have this complex an algorithm for partitioning a database. The algorithm may be useful in the fields of picture segmentation and cluster analysis, however. We believe it is a good region-growing technique and if made more efficient could be very useful in that area.

6.3 Suggestions for further research

The partitioning method deserves much more study. We have shown that it can work in an artificial database with randomly generated records. Further research should try to extend these results to real statistical databases, usually containing many more attributes and having different relation-

ships among the attributes and data fields. Very little research in this area has been performed on actual databases with the notable exception of Schlörer [30]. The need for such protection is real and increasing as more and more personal information is stored in computers.

Dynamic databases provide special problems for partitioned databases as it is the nature of partitions to be static. Changes in the system, particularly inserting and deleting records, affect the pre-formed groups and may require repartitioning. Implementation of an effective dynamic partitioning method is a research challenge which should be investigated.

BIBLIOGRAPHY

[1] Alagar, V.S., Blanchard, B., and Glaser, D. Effective Inference Control Mechanisms for Securing Statistical Databases. AFIPS Joint Computer Conference Proceedings, Vol. 50, 1981.

[2] Bachi, R. and Baron, R. Confidentiality Problems Related to Data Banks. Proceedings of 37th Session of International Statistics Institute, Sept. 1969, p.225-241.

[3] Beck, L.L. A Security Mechanism for Statistical Databases. ACM Trans. Database Syst. 5,3, Sept. 1980, p.316-338.

[4] Blanchard, B. Effective methods for statistical database security, Masters thesis, Department of Computer Science, Concordia University, Jan. 1981.

[5] Chin, F.Y. Security in Statistical Databases for Queries with Small Counts. ACM Transactions on Database Systems 3,1, March 1978, p.92-104.

[6] Conway, R. and Strip, D. Selective Partial Access to a Database. Proceedings of ACM 1976 National Conference, Houston, Texas, Oct. 1976, p.85.

[7] Davida, G.I., Linton, D.J., Szelag, C.R., and Wells, D.L. Database Security. IEEE Transactions on Software Engineering SE-4, #4, Nov. 1978, p.531-533.

[8] Demillo, R.A., Dobkin, D., and Lipton, R.J. Even Databases that Lie can be Compromised. IEEE Trans. on Software Engineering SE-4, #1, Jan. 1978, p.73-75.

[9] Denning, D.E. A Review of Research on Statistical Database Security. In Foundations of Secure Computation, R.A. DeMillo et al. Eds., Academic Press, New York, 1978.

[10] Denning, D.E. Are Statistical Databases Secure? AFIPS Joint Computer Conference Proceedings, Vol. 47, 1978, p.525-530.

[11] Denning, D.E. and Denning, P.J. Data Security. Comput. Surv. 11,3, Sept. 1979, p.227-249.

[12] Denning, D.E., Denning, P.J., and Schwartz, M.D. The Tracker: A Threat to Statistical Database Security. ACM Trans. on Database Systems 4,1, March 1979, p.76-96.

[13] Denning, D.E. and Schlörer, J. A Fast Procedure for Finding a Tracker in a Statistical Database. ACM Trans. Database Syst. 5,1, March 1980, p.88-102.

[14] Denning, D.E. Complexity Results Relating to Statistical Confidentiality. Computer Science and Statistics: 12th Ann. Symp. Interface, Waterloo, Canada, May 1979, p.252-256.

[15] Denning, D.E. Secure Statistical Databases with Random Sample Queries. ACM Trans. Database Syst. 5,3, Sept. 1980, p.291-315.

[16] Dobkin, D., Jones, A.K., and Lipton, R.J. Secure Databases: Protection Against User Influence. ACM Trans. on Database Systems 4,1, March 1979, p.97-106.

[17] Fellegi, I.P. On the Question of Statistical Confidentiality. J. Amer. Statist. Assoc. 67,337, March 1972, p.7-18.

[18] Fellegi, I.P. and Phillips, J.L. Statistical Confidentiality: Some Theory and Applications to Data Dissemination. Annals of Economic Social Measurement 3,2, April 1974, p.399-409.

[19] Hansen, M.H. Insuring Confidentiality of Individual Records in Data Storage and Retrieval for Statistical Purposes. AFIPS Joint Computer Conference Proceedings, Vol. 39, 1921, p.579-585.

[20] Haq, M.I.U. Security in a Statistical Database. Proceedings of American Soc. for Information Sciences, 37th meeting, Vol. 11, Oct. 1974, p.33-39.

[21] Haq, M.I.U. Insuring Individual's Privacy from Statistical Database Users. AFIPS Joint Computer Conference Proc., Vol. 44, 1975, p.941-945.

[22] Haq, M.I.U. On Safeguarding Statistical Disclosure. Proceedings of the International Computer Symposium, Liege, Belgium, 1977, p.491-495.

[23] Hoffman, L.J. and Miller, W.F. Getting a Personal Dossier from a Statistical Data Bank. Datamation 16,5, May 1970, p.74-75.

[24] Kam, J.B. and Ullman, J.D. A Model of Statistical Databases and their Security. ACM Trans. on Database Systems 2,1, March 1977, p.1-10.

[25] Karpinski, R.H. Reply to Hoffman and Shaw, Datamation 16,12, Oct. 1970, p.11.

[26] Nargundkar, M.S. and Saveland, W. Random Rounding to Prevent Statistical Disclosures. Proc. Amer. Stat. Assoc. Soc. Stats. Sec., 1972, p.382-385.

[27] Records, Computers and the Rights of Citizens. By the U.S. Department of Health, Education and Welfare. n.p.: The Colonial Press Inc., 1973.

[28] Reiss, S.B. Medians and Database Security. In Foundations of Secure Computation, R.A. DeMillo et al. Eds., Academic Press, New York, 1978.

[29] Reiss, S.P. Security in Databases: A Combinatorial Study. JACM 26,1, Jan. 1979, p.45-57.

[30] Schlörer, J. Identification and Retrieval of Personal Records from a Statistical Data Bank. Meth. of Information in Medicine 14, 1, Jan. 1975, p.7-13.

[31] Schlörer, J. Confidentiality of Statistical Records: A Threat Monitoring Scheme for On Line Dialogue. Meth. of Inf. in Medicine 15, 1, Jan. 1976, p.36-41.

[32] Schwartz, M.D., Denning, D.E., and Denning, P.J. Linear Queries in Statistical Databases. ACM Trans. Database Syst. 4,2, June 1979, p.156-167.

[33] Yu, C.I. and Chin, F.Y. A Study on the Protection of Statistical Databases. ACM SIGMOD Conference on Management of Data, Toronto, Ontario, Aug. 1977, p.169-181.