DECISION RULES FOR THE AUTOMATED GENERATION OF STORAGE STRATEGIES IN DATA MANAGEMENT SYSTEMS

by

GRANT N SMITH

S.B., MIT
(1974)

SUBMITTED IN PARTIAL FULFILLMENT

OF THE REQUIREMENTS FOR THE

DEGREE OF MASTER OF

SCIENCE

at the

MASSACHUSETTS INSTITUTE OF

TECHNOLOGY

June, 1975

Signature of Author ....
Alfred P. Sloan School of Management, May 9, 1975

Certified by ....
Thesis Supervisor

Accepted by ....
Chairman, Departmental Committee on Graduate Students

DECISION RULES FOR THE AUTOMATED GENERATION OF STORAGE STRATEGIES IN DATA MANAGEMENT SYSTEMS.

by

GRANT N SMITH

Submitted to the Alfred P. Sloan School of Management on May 9, 1975 in partial fulfillment of the requirements for the degree of Master of Science.

Current methods of determining storage strategies (both logical and physical) rely usually on (1) expert opinion, and (2) the experience of the designers. There has been some work in the area of automated design, but the approaches taken to date generally apply only at generation time, thus leaving the resulting design in effect for the rest of the life of the system. Should usage of the system change over time, as experience shows that it will, large inefficiencies may result owing to the original choice of storage strategy.

The work presented here attempts to introduce dynamic decisions regarding storage strategies that will be invoked (1) on a regular basis, and (2) when system performance degrades below an unacceptable level. These decisions involve both the structure of the data base (such as which fields are to be in which files), as well as indexing, data encoding, factoring and virtualizing decisions. Decision rules are described which achieve this result.

Also described is a procedure whereby any given request will be most efficiently satisfied, making use of the current structure of the data base, indexes, etc.

Finally, the set of decision variables required to drive the above decision subsystems is specified in detail.

Thesis Supervisor: Stuart E. Madnick

Title: Assistant Professor of Management Science

I wish to thank Professor Stuart E. Madnick for his invaluable advice, comments and criticism throughout not only the writing of this thesis, but through all the years in which I have been so fortunate as to be associated with him.

In everyone's life, there is one eternal optimist. In mine, that is Professor John Donovan. He has been a powerful motivating force throughout this work.

Last, but by no means least, thanks are due Chip Ziering for many hours of fruitfull - often heated - discussion.

Table of Contents

List of Figures

Introduction.

For some years now the concept of data-independent applications programming has been expounded. What was primarily at stake was the avoidance of rewriting of applications programs if and whenever the underlying data base was changed. Involved was a mapping from the logical data structure (as the data structure available to the applications program came to be known) into some machine oriented data structure (or the physical data structure), the idea being that the system would take care of this mapping function. Then, if there were any change in the physical structure, the mapping function would be changed so that the same logical structure as existed before this change would still be presented to any applications programs, and, in fact, any user (be it in the form of programs, or a person generating requests against that logical data structure). Note that throughout we shall use the term user to mean either a program or a person. It is not necessary for our purposes to distinguish between these two classes, since as far as the database is concerned, all requests look alike.

Arising from this approach is a division of responsibility, and thus of expertise. The logical structure of the data is in the domain of responsibility of the user, while the physical structure and the mapping function previously also in the domain of the user, have now been removed. This is in fact a desirable feature as the user may concentrate efforts on applications-oriented problems rather than becoming bogged down in the technicalities of establishing a data base.

However, that is not quite the way things turned out. There were several attempts to design systems which would perform the mapping function and handle the physical structures for the user, given the logical structure. But the way it turned out was that the mapping capabilities tended to be rather simplistic in concept and execution, with the result that the user had to be quite knowledgable about the physical structure (and thus the mapping function). Furthermore, no differentiation of responsibility was generally delineated and so the user (or user group) now took on the responsibility of both the logical- and physical structures. True, for any one logical structure the user now had a choice of physical structure coming from a wider range than personal experience might previously have allowed, but whether that was a blessing or a disguised horror remains unclear.

Other promises made - and not kept - about data independence again revolve around the mapping function. Theoretically, a given physical structure should be able to be mapped into several logical structures, and vice-versa. This facility has not been realized to any notable extent.

Furthermore, the primary purpose of data independence, namely the isolation of users from changes in the physical data structure, has not yet realized its full potential. Rarely, if ever, was the physical data structure altered once established. It was a Herculean task to implement any one physical structure, and no one was about to go in and tamper once it was working.

Any one physical-to-logical structure mapping would generally be performed only once, and decisions as to what it should be were made at one point in time, with a fixed perception as to the future uses of the data base. These decisions were, and still are, made by <u>people</u>. Much of the knowledge on which these decisions were based was knowledge gained from experience, and so was more akin to an art than a science. However, some non-trivial subset of such decisions are indeed logical and rational, and so subject to some measure of automation.

It would be inaccurate to claim that no attempt has been made to take advantage of the structured nature of some of these decisions. On the contrary, there have been several efforts addressed to this task, and these efforts can be divided (perhaps unfairly) into two major groups:

- simulation-oriented decisions used <u>prior</u> to system generation to aid in structuring decisions. These are notably static, one-time decisions made at the discretion of some person, and requiring substantial human intervention. The results of decisions made at that point were to be influential throughout the life of the data base. However, much credit is due the effort to formalize some major aspects of the decision.
- dynamic rules used continually throughout the life

of the data base to monitor system usage and performance. The results of this monitoring effort would, again, require major human intervention in their interpretation and acting upon. However, the important aspect of these efforts was in that they attempted to track the system on an ongoing basis. Whether any action was taken on these results was questionable. Once again there arose the dilemma of whether to tamper with a working (albeit inefficiently) system.

This work is intended to draw on the invaluable insights gained over the years in dealing with such systems as purport to provide data independence, and some logical-to-physical structure mapping, and to propose a methodology for achieving some of the promises made earlier. It is important to emphasize that this is a methodology since no one work could pretend to be all-encompassing. The approach here will be to:

1) describe a system in which there is true data independence based on a physical-to-logical mapping capability,
2) enhance this system with the ability to perform some of the better formulated decision tasks,

including the monitoring of system use and the dynamic reconfiguration of the physical data structures without alteration of the logical structures. Attention will also be paid to the initial structuring decisions made at definition time.

3) further enhance the system with decision capabilities that are oriented toward the efficient satisfying of requests against the data base.

The work here revolves around the relational model of data. This should not be construed to be a dismissal of all other models (such as the network model) as inferior. The author's familiarity with the relational model and the existence of a well-defined set of theoretical rules that can be applied in the model were the motivating factors behind this decision. It should also be pointed out that the relational model as herein used has embellishments and alterations derived from various personal experiences and sources of the author. The responsibilities for any errors and inconsistencies in the model employed here should not necessarily be attributed to the well-known names behind the relational model; they may well be the fault of the author.

Structure of Thesis.

Chapter II will introduce the relational model as needed for our purposes, and point out the differences, where applicable, between this model and that found in most of the literature.

Chapter III will address itself to the methodology employed for achieving data independence.

Chapter IV presents a list of decision variables maintained by the system. Since there is a long list of statistical information about system usage and performance required to support dynamic decisions regarding physical restructuring, a consistent set of rules has been developed for naming these decision variables.

Chapter V will address itself to the decision rules responsible for initial specification, and subsequent dynamic reconfiguration of the physical data structures - the Structural Decision Subsystem (or SDS), and chapter VI will concern those decisions made dynamically about optimally satisfying requests made against the data base - the <u>Request Decision Subsystem</u> (or RDS).

Chapter VII presents a typical scenario, and those decision rules developed in Chapters V and VI will be applied to the scenario to demonstrate the effectiveness of the decision rules.

Chapter VIII concludes the thesis with some remarks as to further possibilities that can, and perhaps should, be explored, as well as ways to expand the decision rules utilizing a similar methodolgy to that employed here.

Again, it must be pointed out that the decision rules developed in Chapters V and VI are situation specific (and certainly dependent on the implementation of Chapter III) and are clearly not universally applicable. They are intended to demonstrate a methodology and there is no intention of developing a comprehensive and universal set of rules.

Finally, some familiarity with BNF (Backus-Normal Form) is assumed throughtout. Good introductory sources are (1,2).

The Relational Model of Data.

Probably the major stumbling block in introducing the relational model is the terminology. The concepts underlying this approach are familiar to us all.

Consider a regular report, or table that we have all seen at one time or another. In Figure 2.1 is such a table; a convenient format for representing such data. The columns spell out the categories of data; the rows provide a value for each category. Note that the rows and columns might well be interchanged without loss or alteration of meaning. For example, in Figure 2.1 we see the columns labelled 'dept#', 'description', etc. And there are 7 rows. No-one has difficulty in interpreting the information in Figure 2.1, and this is essentially the relational model.

By convention in the relational model, we always label the <u>columns</u>, and put the data in the <u>rows</u> (ie: horizontally) just as is the case in Figure 2.1 . Furthermore, the columns are called <u>domains</u>, and the column headings are thus <u>domain names</u>. This arises from the mathematical concept of a domain

Plant: White Plains, New York.

Period ending: Aug 31.

Summary of Operations

(in 000's)

Dept#	Description	Labor	Expense Actual	Budget	Difference (Actual-Budget)
1	Spray	2990	6464	7103	- 639
2	Coating	5915	12829	13981	-1152
3	Filing	998	2590	2190	+ 400
6	Sanding	1637	3907	5243	-1336
7	Buffing	5915	11275	10750	+ 525
10	Assemble and Pack	4788	8846	8998	- 152
	TOTAL	22243	45911	48265	- 2354

Figure 2.1

as being a collection of objects (or numbers, or any other information-carrying item). When we choose a value from that collection we are choosing an item from that domain.

Notice that each row is created by choosing a single item from each of the six domains. Each row in the table is called an entry. Notice also that the order of the entries (rows) in the table is not important. We might just as easily put the 'total' entry at the top of the table, and then the departments in decreasing 'dept#' order. In fact, we lose no information if we shuffle the rows; it may be inconvenient to have the rows in random order (as it would be, for example, in a telephone directory) but no information is lost by a random ordering of the entries.

Now, if we were to interchange domains 1 and 2 of Figure 2.1 (ie: 'Dept#' and 'description') there would be no problem provided we changed the domain names (column headings) as well. But notice that the order of the domains within any one entry must be the same as that in all other entries if the table is to remain meaningful. Thus, the order of the domains is important, while that of the rows is not.

Primary Keys.

In Figure 2.1 we may observe that there can be only one entry in the table for any one value of 'Dept#', and the same applies for 'description', while there is no reason for this to be the case in any of the other columns. In fact, in the 'labor' domain the value '5915' appears twice. Thus, given the value '5915' and told that it is in the 'labor' column of Figure 2.1, we can not determine from that information alone which department it is that is meant. (If it is both departments, then there is no problem.) But, given a value for 'Dept#', there is no ambiguity about any information relevent to that row. Eg: given Dept# = 2, we can uniquely determine all other values in the entry. Thus, we say that 'Dept#' is a candidate primary key for the table; ie: for any value of 'Dept#' there is only one entry in the table.

In the event that there is no such domain, then some combination of domains must be found that exhibit this property; namely, given a set of values for that comination of domains, the entry containing those values in those domains is uniquely determined.

A table <u>need</u> <u>not</u> have a primary key, but it is often advantageous from the standpoint of efficiency to do so. (In the relational model propsed by Codd, et. al. no relation may contain two entries that are identical, and so there always exists some primary key, even if it is a combination of all domains. This is not the case here, as can be seen in the definition of the 'Join' operator in Appendix 2.)

<u>Normalization</u>.

Looking again at Figure 2.1, we notice that printed above the table is some additional information, such as the 'Plant', and the 'Period' covered. We see also that there are two columns under the heading of 'Expense'; viz. 'Actual' and 'Budget'.

Since the relational model views the world as a set of tables, we must find some way to include that information in the table itself. As it now stands, it is not really part of the information in the table; rather it is a form of table heading. Considering the fact that the 'Plant' and the

'Period' are printed at the top of the table, we may assume that it is of some importance, and we further assume that there are other plants and other periods.

One course of action is to set up a distinct table for each plant/period combination, each having an identical format to that of Figure 2.1 . This would result in a large number of identical, yet distinct tables, and so a second course of action suggests itself: set up a single table for all plant/period combinations, and somehow distinguish entries as belonging to some specific plant and period. This can be done by simply adding two domains to Figure 2.1: 'Plant' and 'Period'.

Furthermore, we must find some way of incorporating the notion of 'Expense' into the two domains 'Actual' and 'Budget'. To do so, we might merely rename the domains 'Actual expense', and 'Budget Expense'. The table now is as appears in Figure 2.2 .

Notice, however, that the table contains two domains each based on the notion of 'Expense'; we have just renamed the

Summary of Operations

(In 000's)

Plant	Period	Dept#	Description	Labor	Actual Expense	Budget Expense	Difference (Actual-Budget)
W Plns	10/31	1	Spray	2990	6464	7103	- 639
W Plns	10/31	2	Coating	5915	12829	13981	-1152
W Plns	10/31	3	Filing	998	2590	2190	+ 400
W Plns	10/31	6	Sanding	1637	3907	5243	-1136
W Plns	10/31	7	Buffing	5915	11275	10750	+ 525
W Plns	10/31	10	Assemble and Pack	4788	8846	8998	- 152
W Plns	10/31		TOTAL	22243	45911	48265	-2354

Figure 2.2

two domains as in Figure 2.2. In this case, the values appearing in either column are, in fact, chosen from a single domain: the 'Expense' domain. The reason for prefixing 'Actual' and 'Budget' to the domain name was to specify the <u>role</u> of each of these domains in the table. In general, if a domain is used more than once in any one table, it must be qualified by a <u>role name</u>. If there is a failure to provide such role names in that event, then ambiguity results.

Use of a role name is not limited to cases in which a domain is used more than once in the same table, and any domain name may be qualified by a role name.

Figure 2.2 is a version of the table which has unique domain (or rather role) names, and is set up in such a way that it contains <u>all</u> information in the table itself as opposed to some of it in the form of table headings. This is called a <u>normalized table</u>. In general, normalizing a table consists of taking information that applies to all entries (such as the plant and period of Figure 2.1) and making it an integral part of the entries themselves (as in Figure 2.2). More specifically, we take the primary key of tables higher up in the hierarchy, and make it part of the primary key of the lower table. An example will help to clarify this point.

The example appearing in Figure 2.3(a) shows the logical view of the data that might exist in a corporate data base. Figure 2.3(b) is <u>one</u> form of a set of tables that might be formed to store this logical view. Notice that some domains (such as 'children' in the 'employee' table) are not really domains, but the <u>names</u> of other tables.

Figure 2.3(c) is the <u>normalized</u> set of tables arising from 2.3(a). Notice that Figure 2.3(c) was derived from Figure 2.3(b) by the following steps:

- for each domain name in a table (say A) that is in fact a table name (say B) take the primary key of table A, and make it part of the primary key of table B.
- remove the table name (B) from table A.

This is the process of normalization, and, in the relational model, <u>all</u> tables must be normalized (ie: must not contain

```
                              EMPLOYEE(EMP.NAME,AGE)
                                    |
                                    |
      ------------------------------|----------------
      |                                              |
      |                                              |
JOBHIST(JOB.DATE,TITLE)               CHILDREN(CHILD.NAME,AGE)
      |
      |
SALARY(SAL.DATE,SALARY)
```

(a)

1) EMPLOYEE(EMP.NAME,AGE,JOBHIST,CHILDREN)
2) JOBHIST(JOB.DATE,TITLE,SALARY)
3) SALARY(SAL.DATE,SALARY)
4) CHILDREN(CHILD.NAME,AGE)

(b)

1) EMPLOYEE(EMP.NAME,AGE)
2) JOBHIST(EMP.NAME,JOB.DATE,TITLE)
3) SALARY(EMP.NAME,JOB.DATE,SAL.DATE,SALARY)
4) CHILDREN(EMP.NAME,CHILD.NAME,AGE)

(c)

Figure 2.3

(Primary keys underlined)

domain names that are in fact table names).
Why 'relational' model? What we have been calling tables are call 'relations' in the relational model. This is more than an arbitrary name. Remember that we described above how an entry is formed by selecting a value from each domain in the table. In mathematical terminology, these entries are a subset of <u>all</u> combinations of values, or a <u>cartesian product</u> of the domains. The name used for such a subset is a 'relation'.

More formally:
The cartesian product of A and B (written A X B) is a set of ordered pairs, each first element of the pair coming from A, and each second element from B.

Ie: $A \; X \; B = |(a,b): a<A, \; b<B|$

('<' means 'is a member of')

We can easily obtain an ordered n-tuple (where n>2) by this method:

$$D1 \; X \; D2 \; X \ldots X \; Dn = |(d1,d2,\ldots dn): di \; Di, \; i=1,\ldots n|$$

A relation will normally be written as a relation name followed by an ordered, parenthesized list of domain names. Ie: R1(D1,D2,D3). For example: Employee(name,emp#,dept#).

The reader is referred to (3) for further discussion of relations and normalization.

Second Normal Form and Functional Dependence.

The process of normalization described above (namely, the removal of all domain names that are in fact relation names) is adequate for most situations in which the user is careful to ensure that the domains asigned to the various relations are in fact assigned to the 'correct' relations. This is aided by the process of diagramming the data base as shown in Figure 2.3(a). However, there are times when what seems quite logical will, in fact, give rise to problems.

Consider Figure 2.1 . Notice that for any given value of 'Dept#' the value of 'description' is uniquely determined; or in other words, 'description' is functionally dependent on 'Dept#'. Clearly, in this case, the reverse is true as well; namely, 'Dept#' is functionally dependent on 'description', but this need not be the case.

Now, if for any reason we were no longer interested in Dept# 2 and therefor struck the second row from Figure 2.1, we lose the fact that Dept# 2 is 'coating'; ie: the relationship (2,coating) does not exist anywhere else. One way to avoid this is to establish a new relation containing only the functionally dependent domains (Dept#, description). We may now strike either of these domains from the relation in Figure 2.1 without loss of information.

These relations are said to be in second normal form; Ie: Domains not functionally dependent on each candidate key are stored in a separate relation. Figure 2.1 would thus contain 'Dept#' as a domain, but not 'description', and another relation now contains 'Dept#' and 'description'.

Third Normal Form and transitive dependence.

Third normalized relations are second normalized relations in which there exist no transitive dependencies.
If B is functionally dependent on A, and C is functionally

dependent on B, then by the algebraic transitivity laws, C is also dependent on A. But in a somwhat different manner, since it is also dependent on B which is dependent on A. In this case, we say that C is transitively dependent on A. This is true in any case where the application of algebraic transitivity yields an additional functional dependency, as it did in the above case.

Relations in third normal form would not contain any domains that were dependent on any other domain which is itself functionally dependent on some domain in the relation.

For the case above, where C is transitively dependent on A, we would establish a separate relation containing domains B and C, and remove C from the relation containing A.

It thus appears preferable to retain all relations in <u>third normal form</u> for the reasons outlined above.

The reader is referred to (4) for a more comprehensive treatment of second- and third normal forms.

<u>Transferability of Role Names</u>.

Consider the existence of two relations:

person(soc_sec, name, age), and

marriage(husband.soc_sec, wife.soc_sec) .

Notice that 'person' contains the domain 'soc_sec' and so does 'marriage'. Since 'marriage' contains that domain twice, a role name is mandatory. Those supplied are: 'husband.soc_sec' and 'wife.soc_sec'. Now consider a request to list the name and age of the wife of a person with soc_sec 617-03-2911. This might be phrased (in some arbitrary retrieval language) as follows:

list wife.name and wife.age for husband.soc_sec '617-03-2911' ;

Notice that the 'person' relation contains the domains referenced (viz: 'name' and 'age') but not the role name qualifier 'wife'. Intuitively, however, it is clear that the information needed to satisfy the request is present, but not in any way that the system can utilize.

The way that the system makes use of implicit information of the type in the example above is by <u>transferring</u> the role name qualifier 'wife' to the 'person' relation <u>only</u> for that entry designated by the relationship between 'husband.soc_sec 617-03-2911' and the corresponding 'wife.soc_sec'. 'wife' does <u>not</u> become a permanent role name qualifier in the 'person' relation.

<u>Set Theoretic Notation, Definitions and Examples</u>.

In chapter I was mentioned the fact that a well-defined collection of theoretical rules exists which may be used to operate on relations (regardless of whether they are in any particular normalized form). This section outlines these rules. This is perhaps where the model used here differs most from those presented elsewhere(3,4). Differences will be pointed out at appropriate points in the discussion.

The following operations will be defined:

Operation	Symbol	Diadic/monadic **
Union	U	Diadic
Intersection	N	Diadic
Difference	-	Diadic
Cartesian Product	X	Diadic
Projection	P	Monadic
Join	*	Diadic
Composition	.	Diadic
Permutation	M	Monadic
Compaction	C	Monadic
Restriction	R	Both
Division	/	Diadic

These operations are briefly described here, and are formally defined and examples given in Appendix 2.

** Diadic operators operate on two relations (they may both be the same relation); monadic operators operate on a single relation.

Notation.

R<i> is the name of the i th relation

∈ means 'is a member of'

|....| implies a *list*, or *set* of the items between the '|'s.

c(i) is the cardinality (number of entries) in R<i>

n(i) is the degree (number of domains) in R<i>

d(i,j) is the j th domain of R<i>, j=1,..n(i)

v(m)(i,j) is the m th value of d(i,j), m=1,..c(i)

t(i) is an n(i)-tuple in R<i>

ie: t(i) (v(a)(i,1),v(a)(i,2),...v(a)(i,n(i)))

a 1,...c(i)

L(|a|) is the length of list a

∅ is the null set - ie: R<i>=∅ implies c(i)=0

a⊆b means a is a subset of b (a=b is legal)

a⊂b means a is a *proper* subset of b (a≠b)

∀a means for all values of a

This notation will be used throughout the remainder of this thesis.

Explanation of Operators.

Union

The union of two sets consists of a set that contains all entries that appear in either of the two sets.

Intersection

The intersection of two sets is a set that contains only entries that appear in both of the two sets.

Difference

The difference of two sets is a set that contains all entries that appear in one of the sets, but not in the other. Eg: If the two sets were A and B, then 'A - B' is a set of all entries that appear in A, but not in B.

Cartesian Product

This is as defined on Page 25

Projection

The projection of a relation is a procedure whereby some of the domains in the relation are removed.

Join

A join of two relations is the process whereby two relations may be combined into a single relation containing all the domains of the two being joined.

Composition

This is the same as the join, except that the domain on which the relations are joined is removed. This means that a composition is in fact, a projection of a join.

Permutation

A permutation applied to a relation consists of merely re-ordering the domains in the relation.

Compaction.

The compaction operator is used for deleting all redundent entries from a relation. It is used most commonly in conjunction with the projection operator, which may result in redundent entries.

Restriction

The restriction operator is used for selective retrieval from a relation.

Division

Division is the inverse of the cartesian product.

Introduction to XRM.

This section is intended to be a very brief introduction to the pertinent points about XRM.

XRM is a particular implementation of an n-ary relation data management model designed and built by IBM Scientific Center, San Jose (5). It operates basically as follows.

XRM can handle two types of information:

- character string data, and
- fullword (32 bit) numeric data

There are correspondingly two major subcategories of relations; one that handles character strings, and another that handles n-tuples of numeric data. Any one relation type (character or numeric n-tuple) can <u>only</u> handle data of that type.

Each entity in the system (character string, or n-tuple) is automatically assigned an XRM ID when entered into the data base. Given that ID, the entity can be rapidly and efficiently retrieved by XRM. And, given the entity, XRM obtains its ID by applying a hashing function to that entity, and then performing the retrieval. In the case of character strings, some number of the first bytes of the string are hashed; in the case of numeric n-tuples, <u>all</u> primary key domains are hashed.
All IDs in XRM are fullword integers.

In numeric relations (n-tuples), any domain can be <u>inverted</u>.

This is equivalent to building an index for that domain. Once such an inversion exists, given a value for that domain, XRM will rapidly find all ID's of n-tuples in the relation that contain the given value in the inverted domain. If no inversion existed, a linear search would be necessary. More is said about the implementation of inversions in Chapter V.

For our purposes, this introduction will suffice. Additional concepts will be explained as needed. For further information, the reader is referred to (5).

__Shared Data Bases and User Flexibility.__

This chapter presents a methodology based on the relational model for achieving independence between the logical- and physical data structures.

One of the intentions of data independence is to allow the user to view the structure existing in the data he (it) uses in a way most convenient for a specific application. This means that the user should be provided with the facility to define any relation containing any domains in any order, and be able to use it as such. Notice, however, some of the issues raised by permitting this flexibility.

The most glaring problem arises as a result of the divergence from the concept of __shared__ (or centralized) data bases. The benefits of shared data banks are many and have been adequately covered elsewhere. Now we are proposing the facility for allowing every user a powerful tool that allows rapid and easy definition of relations for specific applications. What does this do to the centralized data base concept? Each user now wants (and is able to have) different relations for his application(s), which is basically gaining efficiency and convenience at the expense of generality. Each user must, furthermore, collect and maintain his own data needed to support his application, rather than delegate that function to a central authority.

The traditional method of centralizing data collection and maintainence has been the appointment of a data base administrator whose responsibility it is to maintain the central shared data base, and ensure that all users conform to that data base. Change to the data base is expensive and time consuming, and so generally to be avoided. User convenience was sacrificed in favor of a centralized data base.

There is no need for sacrifice on either the user's part, or the data base administrator's part. This is where the concept of a 1:n mapping of physical to logical structures demonstrates its value. There is no reason for denying a user a specific mapping from the single physical data base into a specific logical relation for some application. This presents no problem if the logical relation that the user wishes to define on the physical data base is some subset of the domains existing in that physical data base. But what if the logical relation requires a mapping onto a domain that does not yet exist in the physical data base, and is yet to be created?

One possibility is to redefine the existing relevent relation in the physical data base to include the new domain. Alternatively one could invoke the principle of a 1:n mapping, now <u>from</u> the logical <u>to</u> the physical relations, and create a new relation containing the required information.

We have thus expressed the need for a n:m mapping from logical to physical relations; ie: a logical relation can map onto several physical relations, and a physical relation can be mapped into several logical relations.

Before proposing a methodology for implementing n:m mappings, let us address very briefly the issue of efficiency. In a very large data base, a user that constantly uses the same, small subset of data in a logical relation pays a high price in performing the mapping each time. Some exception should be made in such a case whereby a physical relation is established containing that subset of data, and existing alongside the original physical relations. This should not however be made to appear any different to the user; the logical relation defined must still appear to be the same and contain fully updated information.

We turn now to a methodology.

Methodology.

There are basically three categories of relations that we have expressed a desire for in the above discussion:

- physical relations in the physical (centralized) data base,
- logical (user defined) relations, and
- special physical efficiency-oriented relations.

The terminology to be used here is as follows (and intended to be consistent with the current terminology found in the literature):

- real relations - those relations that exist physically in the data base
- virtual relations - user-defined (logical) relations which are mapped by the system onto the real relations
- derived relations - real (physical) relations that are subsets of the real relations constituting the data base. They exist primarily for efficiency reasons.

We thus have a basic system as shown in Figure 3.1 . Notice that the elements of the system shown in Figure 3.1 interact in a specific way; more precisely, they form a hierarchy. Figure 3.1 can be easily reformatted to yield Figure 3.2. The same is true of all other figures in this chapter: they can be expressed in an hierarchical relationship.

Notice also that this system has not eliminated either the data base administrator or the need for some person (perhaps again the data base administrator) to specify the initial real relations. The features of the system thus far are:

- the ability to define virtual relations on the system maintained real relations, and have the

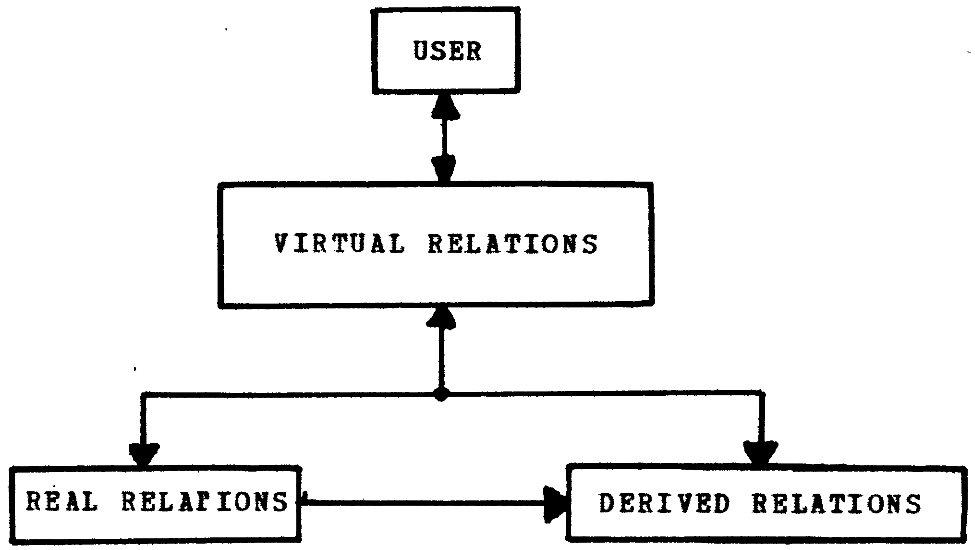

Figure 3.1

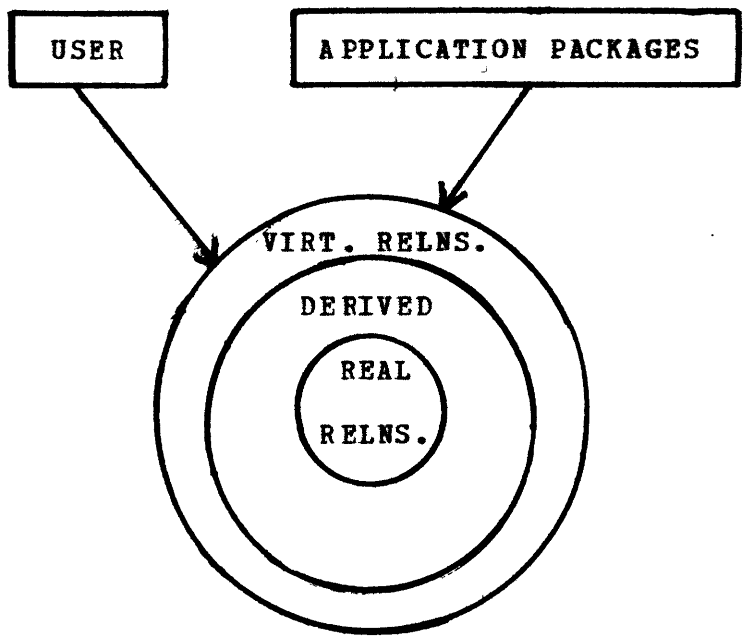

Figure 3.2

system perform mapping functions (note that a virtual relation may in fact be identical to a real relation). More than one virtual relation may be defined on any one real relation.

- the ability to decide (the decision being made by the data base administrator) to create a (real) derived relation for reasons of efficiency in a particular application
- the ability for a virtual relation to contain domains from more than one real relation.

As can be seen, the enhancements are concerned only with the system mapping functions. Thus, users may define virtual relations consisting of domains in any (combination of) real relations, but may <u>not</u> define additional domains. It is also important to point out the following:

- primary keys in virtual relations exist in the eye of the user only; they do not necessarily correspond to primary keys in real relations.
- the set-theoretic operators defined in Chapter II are <u>all</u> that are required by the user for the creation of virtual relations, as virtual relations are a function only of existing relations. The data base adminstrator, however, who needs the capability

to define real relations and/or domains needs additional facilities; perhaps in the form of a DEFINE... or CREATE... command, not available to the user.

Assuming a user wishes to define new real domains, these additional real domains will have to exist as real domains in some real relation somewhere in the data base and there is a decision required as to where in the data base this new domain will exist. Thus adding real domains (or real relations) involves the user interacting with the data base adminstrator. The actions of the data base administrator in this situation would consist basically of the following steps:

1) Determine from the user whether he is merely utilizing a different name for some existing real domain. If so, simply tell the user to define a synonym equating the two names.
2) If (1) is not the case, apply some set of rules to determine in which real relation the domain(s) belong(s).
3) Add the domain to that real relation, thus making it available for use in any user-defined virtual relation. (Note that this step may require

restructuring some real relation. Alternatively, a new real relation could be established consisting of the new domain, and the primary key of the real relation that should contain that domain. A join is required each time the new domain is used. This decision must be made by the data base administrator.)

Step (1) above appears to require some human intervention on the part of a person such as the data base adminstrator, who is familiar with the global system and the existing real relations. But major portions of steps (2) and (3) can indeed be formalized, and automated.

Provided there are some guidelines for the maintaining of real relations (eg: they must all be maintained in third-normal form - see Chapter II), then step (2) above can be performed by the system.

In a similar way, by supplying some information as to the expected use to be made of this new domain, the system can determine precisely how to include this new real domain in the data base - ie: perform step (3). Notice also that this decision is directly analogous to that required in the creation of a derived relation.

•

Perhaps it would appear that all that is accomplished by the automation of the major share of steps (2) and (3) above is the reduction of some administrative overhead. But consider the capability of applying step (3) dynamically, which the data base adminstrator does not have (except perhaps at predetermined, discrete time intervals). This means that the real relations can be so structured as to reflect the current system usage and requirements. Furthermore, because of the n:m mapping capability of the system, these changes in the real relations - be it mere addition of a real domain or relation, or a restructuring of existing real relations - are not visible in any way to the virtual relations of the user. We may now modify Figure 3.1 to show the fact that there is some system function controlling the structure of the real relations in the data base; namely, the Structural Decision Subsystem (SDS). The modified version of Figure 3.1 appears in Figure 3.3 . If Figure 3.3 were reformatted into an hierarchical diagram, the SDS, which must be available to the real relation handlers, would become the innermost level of the hierarchy.

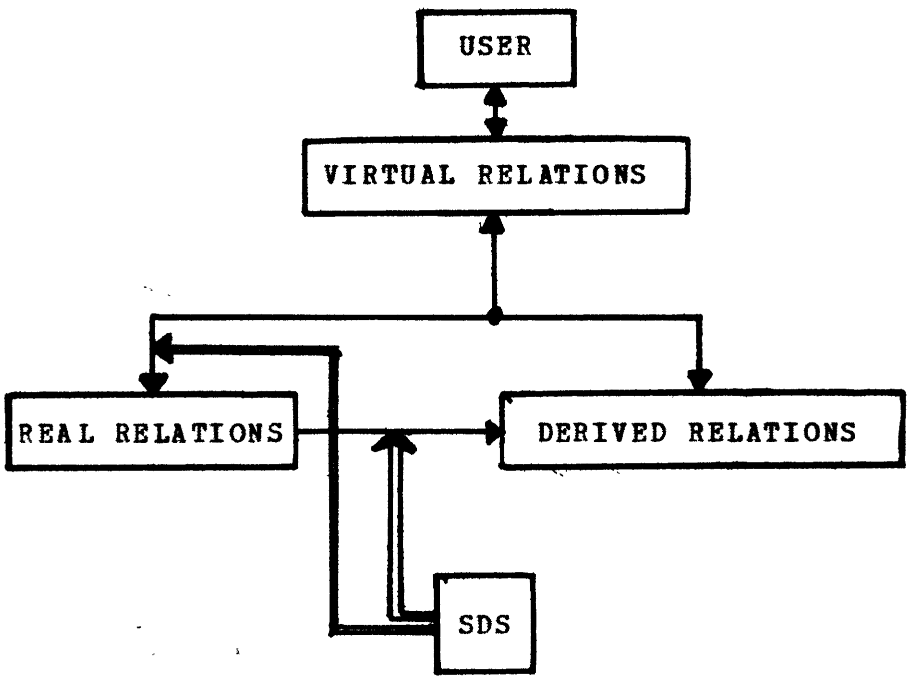
USER
VIRTUAL RELATIONS
REAL RELATIONS
DERIVED RELATIONS
SDS

Figure 3.3

We have discussed thus far in a non-technical manner a methodology for automating some of the functions revolving around the maintainence of the real relations of the data base.

Now, given a structure for the real relations of the data base (as specified by the SDS), and given also the possible existence of derived relations (also determined by the SDS) it becomes clear that there may well be more than one way to satisfy a request against the data base. All requests from users are against virtual relations (or some set-theoretic derivation of virtual relations), which from above, are mapped onto one or more real, or derived relations. Once again some decisions are required in the mapping function to determine:

- whether the request is valid - ie: can logically (and legally, from an access control point of view) be satisfied given the virtual relations involved,
- how the request can be satisfied, and
- how best to satisfy the request.

The subsystem that controls these decisions is the Request Decision Subsystem (RDS). The RDS is responsible also for determining how well it is doing in terms of efficiency. If the RDS decides that system performance is degrading (perhaps as a result of changing system usage) it will trigger the SDS in an attempt to restore performance to an acceptable level. We thus modify Figure 3.3 to include the RDS, as shown in Figure 3.4 . Its position in the corresponding hierarchical diagram is self-evident from this figure.

The discussion in this chapter has purposely been non-technical in nature in an attampt to demonstrate the global functions and interactions within the system of the major decision subsystems - the SDS and RDS. Furthermore, the techniques employed in implementing both the real- and virtual relations are of no consequence to the discussion, and have no impact on the methodology proposed.

Finally, notice that the real- and virtual relations are identical in their conceptual underpinnings, and thus requests against either are made in a consistent fashion. The requests used throughout will be in the format of set-theoretic operations on relations, be they real-, virtual-, or derived relations. (These operations are as

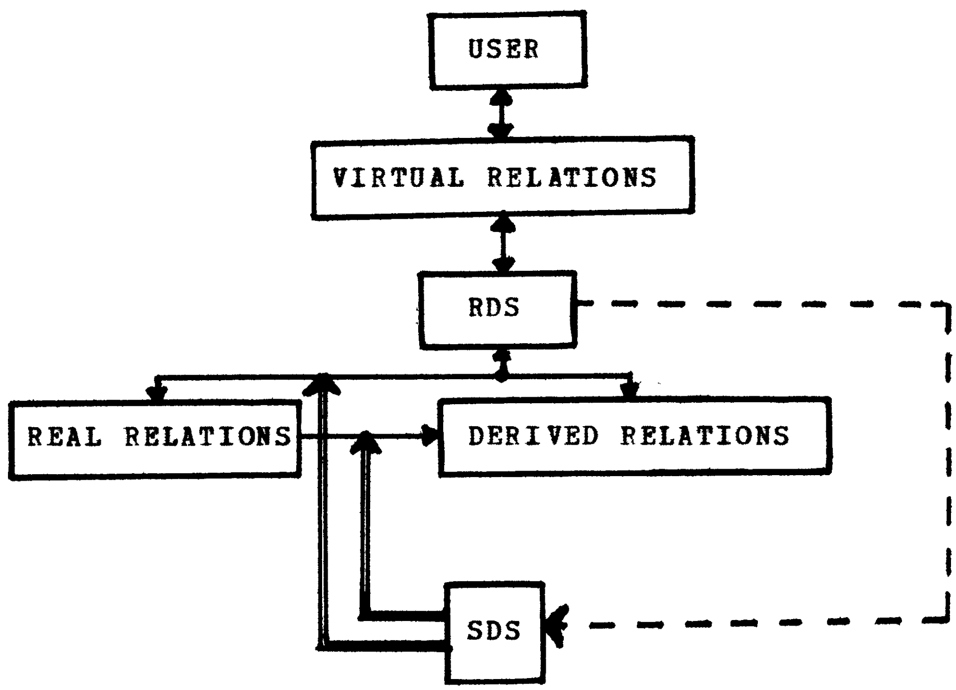
USER
VIRTUAL RELATIONS
RDS
REAL RELATIONS
DERIVED RELATIONS
SDS

Figure 3.4

defined in Chapter II.) This does not of necessity imply that a user will employ set-theoretic requests directly; a mapping from a higher-level request language to a set-theoretic algebra is a well-understood, and conceptually simple operation. (See (6)) Thus our hierarchical view of Figure 3.4 might involve an additional layer between the user and the virtual relations; namely, a request language - to - set theoretic operation mapping facility.

What we have presented in this chapter is a methodology for achieving true data independence and providing the user with powerful facilities for defining application-specific relations. At the same time, however, we preserve the centralized data base concept. The methodology is enhanced by two decision subsystems which assume some of the system structuring responsibility.

We proceed now to a detailed inspection of the decision subsystems.

Decision Variables and Truth Functions.

This chapter describes the naming conventions to be used in subsequent sections for the naming of decision variables.

Since there is a rather large set of these decision variables, it was decided to establish a consistent method for naming them. This method is presented here in BNF (Backus-Normal Form) format, along with the appropriate explanations.

Note that all decision variable names begin with a '$<number>'. This signifies the BNF rule number used to generate that name. A reference section containing these numbered rules appears as Appendix 3.

<relation id> is an XRM-assigned internal ID; <domain #> is the position of a domain within a tuple.

Rule# Rule

1 <qualifier>::= $1<relation type> <unit> <request>
<category> <qualifier type> <join info>
<options>

These are variables containing statictics about the use of domains in the capacity of qualifiers in the list of selection criteria that appear in a request.

<relation type>::= <virtual>|<real>|<derived>

<virtual> ::= v

<real> ::= r

<derived> ::= d

<unit>::=<domain>|<relation>

<domain> ::= d(<relation id>,<domain#>)

<relation> ::= r(<relation id>)

<request> ::= <retrieve>|<update>|<insert>|<delete>

<retrieve> ::= r

<update> ::= u

<insert> ::= i

<delete> ::= d

<category>::=<simple>|<compound>|<non-specific>

<simple> ::= s

<compound> ::= c

<non-specific> ::= n

<qualifier type>::=<equality>|<nonequality> |
<unspecified>

<equality> ::= e

<nonequality> ::= n

<unspecified> ::= u

<join info>::=<join>|<nojoin>

<join> ::= j

<nojoin> ::= n

<options>::=<null>|<index>|<no index>

<null> ::=

<index> ::= i (trss) (trss='total resloved set size')

<no index> ::= n

<u>Example</u>: $1rd(i,j)rsen is the number of times the j-th domain of real relation i is used as a simple (ie: the only) qualifier in a retrieval request, and was used as an equality constraint. No join was needed to satisfy the request.

The <relation type>, <unit> and <request> should be self-evident. For <category>, if there is only one domain in the list of selection criteria, then the <category> is 's'. In the event that there are several domains in the qualifier the <category> is 'c', and if there is a sequential retrieval from the relation, the <category> will be 'n' (or non-specific).

For <qualifier type>, a constraint in a qualifier can be of essentially two types:

- an equality constraint, such as 'age=26',
- an inequality constraint, such as 'income > 20,000'.

If there is no constraint (as is the case when <category> is 'n') then the <qualifier type> is 'u' - or unspecified.

<join info> will be a 'j' in the event that there was a join required in the resloving of the request, and it will be 'n' if no join was necessary.

<options> will be null in the event that <category> is 's'. If <category> is 'c', however, then <options> will show whether some <u>other</u> domain in the list of qualifiers had n inversion in the real relation. If so, then <options> is 'i' - or 'index', and the system will also store the total size of the resolved set (the set of entries that results when those domains with indexes are used first in a restriction). If there was no other domain in the list of domains in the selection criteria, then <options> is 'n'.

<u>Rule#</u> <u>Rule</u>

2 <retrieved object>::= <relation type> <unit> <request> <object> <join info>

This is a set of variables that will contain statistics as to the use of domains as the <u>objects</u> of a request.

<relation type>::=<real>|<virtual>|<derived>

<real> ::= r

<virtual> ::= v

<derived> ::= d

<unit>::=<domain>|<relation>

<domain>::= d(<relation id>,<domain #>)

<relation> ::= r(<relation id>)

<request>::=<retrieve>|<update>|<insert>|<delete>

<retrieve> ::= r

<update> ::= u

<insert> ::= i

<delete> ::= d

<object>::= <simple> | <compund> | <entry> | (<aggregate>)<object>

<simple> ::= s

<compound> ::= c

<entry> ::= e

<aggregate>::=SUM|AV|MAX|MIN|COUNT|UNIQUE

<join info>::=<join>|<nojoin>

<join> ::= j

<nojoin> ::= n

Example 1) $2rr(i)rej is the number of times a whole entry is retrieved from real relation i, when a join was necessary to resolve the request.

2) $2vd(i,j)r(AV)sn is the number of times that the average of the values in only (since <object> is 's') domain j of virtual relation i is retrieved; no joins were needed to resolve the request.

<object> in this rule is similar to <category> of rule #1. The value of <object> will be s if this is the only domain specified for retrieval (or update) in the request. If there are several domains specified, then <object> is 'c'. <object> may also be an <aggregate> if the individual items were not required, but some aggregation of them was.

Rule# Rule

3 <joins>::= $3 <relation type> <domain> <domain>

This type of variable maintains statistics about the involvement of relations in joins. It specifies the number of times a relation was joined to some other relation by a specific domain.

<relation type> ::= <virtual>|<real>|<derived>

<virtual> ::= v

<real> ::= r

<derived> ::= d

<domain>::= d(<relation id>,<domain #>)

Example: $3rd(i,j)d(k,m) is the number of times that relation i was joined by domain j to domain m of relation k.

Rule# Rule

4 <system data>::=$4<system variable>

These variables store information about system parameters and costs.

<system variable>::=<block size> | <index blocking factor> | <cost-per-I/O> | <relation blocking factor> | <cost/byte/day> | <overhead per call to XRM> | <time period>

<block size>::= p

<index blocking factor> ::= bfx(<domain>)

<domain>::=<relation id>,<domain #>

<relation blocking factor> ::= bfe(<relation id>)

<cost-per-I/O>::= io

<cost/byte/day>::= sc

<overhead per call to XRM>::= opc

<time period>::= t

In XRM, bfx(i,j) is constant ∀i,j, and bfe(k) is constant ∀k. So, for our purposes we can refer to them simply as 'bfx' and 'bfe'. The <time period> 't' will be the length of time since the last restructuring of the data base by the SDS. All SDS decisions are based on the period since the last restructuring occurred, and so decisions will be based on this time period.

<u>Rule#</u> <u>Rule</u>

5 <relation data>::=$5<relation variable>

These variables are used to store information regarding relations.

<relation variable>::=<degree><type> |
<cardinality><type>
<degree>::= #d(<relation id>)
<cardinality>::= cy(<relation id>)
<type>::=<real>|<virtual>|<derived>
<real>::= r
<virtual>::= v
<derived>::= d(<method of derivation>)

<degree> is the number of domains in the relation;

<cardinality> is the number of entries in the relation.

<u>Rule#</u> <u>Rule</u>

6 <domain data>::=$6(<domain name>)<domain variable>

These variables are used for maintaining statistics about domains.

<domain variable>::= <# unique values>

<# unique values> ::= q

<u>Example</u> $6(state)q is the number of unique values that will be found in domain 'state'. Notice that for domains that are numeric, $6(i)q is the same as the cardinality of the relation in which domain i appears. For character strings, it may be anything from 1 to the cardinality of the relation in which it appears.

<u>Rule#</u> <u>Rule</u>

7 <user information>::=$7<user variable>

<user variable>::=<response time weight factor>

<response time weight factor>::=r

The <response time weight factor> is a user-supplied preference for how the response time is to be weighted in structuring decisions. It is a value from 0 thru 1 inclusive. For purposes of this thesis, the value of $7r will be 0.5, which is essentially a null value. However, the variable may be taken into account by merely appending '$7r' to all cases where '$4io' and '$4opc' appear in the decision rules, and by appending '(1-$7r)' to all instances of '$4sc' in the decision rules.

Truth functions.

In addition to the statistical variables above, these is a set of truth functions used to test for specific conditions. The names of these truth functions all begin with: '$8'. The value of a truth function is '1' if applying the function to a specific case is true; otherwise the value is '0'. For example, if T(i) were a truth function that tests for negativity, then if i<0, T(i)=1, otherwise T(i)=0.

The truth functions employed here are presented below. (Note that the <type> of a relation (real, derived or virtual) is not important in applying truth functions.)

$8d(i,j) domain j appears in relation i

$8i(j,k) domain k in relation j is inverted. (For virtual relations, $8i(j,k)=0 always.)

$8p(i,j) domain j is one of the primary key domains of relation i.

$8x(i)r relation i is a real relation.

$8x(i)d(<method>) relation i is a derived relation, and <method> is the method of derivation. If <method> did <u>not</u> involve a restriction, then <method>::=<null>.

$8n(i,j) domain j of relation i is mandatory. Ie: a value must be provided for this domain before an entry in relation i will be made.

<u>Note</u> $8n(i,j)=1 ∀j where $8p(i,j)=1. (Primary key domains are mandatory.)

$8u(j) domain j contains unique values (eg: soc_sec_#)

$8r(i,j) same as $8n(i,j) except that it refers to a role name. Also notice that $8r(i,j) is a subset of $8d(i,j) Thus this is a truth function that tests whether a role name is in relation i.

$8_(j)<data type><storage strategy>

<data type>::=<character> | <fixed> | <float> | <vector> | <bit>

<character>::= c

<fixed>::= x

<float>::= f

<vector>::= t(<size>)

<bit>::= b

<storage strategy>::=<virtual> | <real encoded> | <real unencoded>

<virtual>::= v

<real encoded>::= e

<real unencoded>::= u

This set of truth functions is to test the data type of domain j. For exmaple, if $8_(name)ce=1 then domain 'name' is an encoded character string.

$8f(|m|,|n|) is a truth function that tests whether <u>each</u> of the domains in list |n| are functionally dependent on the <u>whole</u> list |m|.

Note 1) List |m| is <u>not</u> a list of <u>all</u> domains on which members of list |n| are functionally dependent. Each n'<|n| may be functionally dependent on some |x|≠|m| <u>also</u>.

2) If |n|=Ø (ie: is empty) then $8f(|m|,|n|)=0.

$8m(|p|,|q|) is a function that tests whether lists |p| and |q| are <u>mutually</u> dependent. Ie:

$8f(|p|,PqP)=$8f(|q|,|p|)=1, and also $8m(|p|,|q|) implies $8m(|q|,|p|).

Transitivity also holds: $8m(|p|,|q|)=$8m(|q|,|s|)=1 implies that $8m(|p|,|s|)=1.

$8c(|p|,q)(<function>) is a function which tests whether q (note that q is not a list) is <u>computationally dependent</u> on domains |p|. For example, if domain q is defined as 'q=6.3 * p' then q is computationally dependent on p. (<function>) is the computation required to derive q from the list of domains |p|.

$8od(k) is a truth function set up for a request. It is '1' if domain k appears as one of the object domains in the request.

$8eq(k) is a truth function used in requests. It is '1' if domain k appears as a qualifier with <qualifier type> 'e'.

$8nq(k) is similar to $8eq(k) except that the <qualifier type> is <u>not</u> 'e'.

Note that truth functions may be implemented as unary and binary relations (depending on the particular truth function) where existence of an entry in the relation signifies 'true', or '1'.

The complete list of decision variables and truth functions that are used in this collection of decision rules is listed in Appendix 3.

In addition, the following notation will be employed in subsequent chapters:

- an '*' appearing in any decision variable name means the <u>sum</u> of all the possible replacements of the '*'.

  For example:

  $1vd(i,j)*sej = $1vd(i,j)rsej + $1vd(i,j)dsej + $1vd(i,j)usej

  ($1vd(i,j)isej is not used.)

  Or:

  $1rd(i,*)rsej = SUM($1rd(i,k)rsej) for k=1,...$5#d(i)

- a '¢' in a name means the <u>product</u> of all possible replacements for the '¢'.

  For example:

  $1vd(i,j)rse¢ = $1vl(i,j)rsen.$1vd(i,j)rsej

- a list between two '|' (vertical bars) means the sum of that list.

  For example:

  $1vd(i,j)|d,u|sej = $1vd(i,j)dsej + $1vd(i,j)usej

The reader is advised to become familiar with the 7 rules and the various truth functions to avoid continual reference to Appendix 3 and thus to expedite reading.

The Structural Decision Subsystem (SDS).

This chapter presents a detailed exposition of the SDS.

The SDS is charged with the responsibility for:

- maintaining the database in third normal form,
- structuring the real relations in such a way that current system usage is most efficiently serviced,
- modifying any system descriptor tables to reflect any change in the structure of the real relations.

Since we are not concerned specifically with any implentation here, we will not address the modification of system descriptor tables. This is, nevertheless, a SDS function.

As outlined in Chapter III, the creation of real relations is a privelaged operation. While any user may define an indefinite number of virtual relations, the disorderly or random definition of real relations would ultimately destroy the centralized nature, and cohesiveness of the data base.

The SDS is concerned only with real relation restructuring since virtual relations may, by definition, only be altered by the user that defined them. However, the SDS must examine virtual relation use for purposes of making decisions about derived relations.

There are two separate points at which the SDS can be invoked:

. at definition time, and

. during the life of the system - or dynamically.

In either case, the function of the SDS is identical; namely, to 'best' structure the database for its expected use. The distinction between these two ocasions of SDS use is simply one of the source of values for the decision variables. At definition time, values for decision variables (and the definition of truth functions) are exogenous, and supplied to the SDS. At any time thereafter however, continuous monitoring provides accurate records of actual use, which become the values for the decision variables at the time the SDS is invoked.

It is important to note that the operation of the SDS is in no way dependent on the source of the decision variable values. Given a set of values, the SDS can operate. Thus, the stage of system life (definition, or subsequent thereto) does not predetermine any particular operations to be performed by the SDS that are not required at other stages.

The SDS presented here is cost centered. Ie: it attempts at all times to minimize cost as opposed, for example, response time. However, in light of the fact that response time is often the most crucial factor for many users, there may be a <response time weight factor> provided by the user ($7r). If none is supplied, the defalut is 0.5 (which is in effect, null). All decision rules here assume that $7r=0.5 .

We proceed now to the SDS proper.

5.1 Maintaining third normal form.

The algorithms within the SDS for maintaining real relations in third normal form are driven by a set of truth functions of the variety presented in Chapter IV.

Every domain _must_ appear either as a functionally dependent domain in some truth function, or some domain on which others are functionally dependent.

It is the responsibility of the user (perhaps in co-operation with the data base administrator) to define these truth functions. The system is not (and in fact no system can be) able to third normalize without substantial user-provided information. In order to do so, the user must understand the interrelations, and peculiarities of the data, and the data base administrator is responsible for education in this function. It is envisioned that the user will employ network-like diagrams to aid in this task (See the scenario of Chapter VII).

Note that the user is _not_ asked to provide third normalized relation definitions per se, but rather the information that will enable the _system_ to define third normalized relations. This distinction is important if one considers the (possible) dynamic nature of the real relations. If a new real domain is to be added to the data base, a decision is required as to which real relation it belongs in. Given the knowledge that the data base administrator has of the data base, he is the clear candidate for making the decision, and he would simply relefine and restructure the affected relation. Notice that this is the <u>only</u> course open to the data base administrator, whereas the system, provided with adequate information would be in the position to <u>dynamically</u> consider alternatives to the full, and expensive, restructuring of the affected relation.

It is thus deemed preferable to provide the necessary information, and to allow the SDS to third normalize in order that dynamic modifications to relations be efficient. The algorithm for third normalization is detailed in Appendix 1. Suffice it to say here that, given a set of functional- and mutual dependencies, the system can generate a database (real relations) in third normal form. Note that computational dependencies are not considered when third-normalizing.

Now, assume that some new real domain is to be added to the database. By having the functional- and mutual dependencies for the new domain, the system can determine where in the set of real relations, this new domain belongs if third normal form is to be preserved. Furthermore, it is able to determine the most efficient method of including it in the data base.

We proceed now to decisions made by the SDS under the assumption that third normalization has occurred, and resulted in a set of real relations of the following type:

<name>(<list of domains>)(<list of candidate keys>)

The primary key will become the <candidate key> with the fewest domains. This maximizes the chances of having values specified for all primary key domains in selection criteria.

For example: RR1(A,B,C,D,E)((A,B),(C,D,E))

The primary key that would be chosen here is (A,B)

5.2 Structuring Decisions.

These decisions are basically those that determine the implementation of the relations specified by the third normalization process. These decisions are:

1) Encoding or virtualizing of domains
2) Indexing decisions (the creation of inversions)
3) Factoring decisions
4) Decisions to join permanently into a single relation

any two relations that have the same primary key.

One of the structuring decisions considered, and subsequently dismissed was that of replacing a relation (in third normal form) by two or more of its projections. The factors that are involved in any such decision are:

a) the cost of transporting little-used domains of a relation to primary memory each time any part of the relation is used (A case <u>for</u> splitting up the relation)

b) the overhead involved in maintaining an extra relation, and duplicate copies of some domains (A case <u>against</u> splitting)

c) The overhead involved in performing a join each time one of the domains split off is required for any reason (A case <u>against</u> splitting the relation)

d) <u>Possible</u> reduced storage (A marginal case for splitting the relation up)

However, since the cost of transporting unneeded domains to primary memory (once the entry has been located, and an I/O is required anyway) is so minimal that it will be clearly dominated by costs of (b) and (c). (d) is an uncertain value. There are occasions in which the cardinality of a projection may be smaller than that of the original relation, but this is never certain.

As such, it appears that the decision to replace a relation by two or more of its projections will never be made, and so was not included in the SDS.

5.3 Encoding and virtualizing decisions.

5.3.1 Virtualizing Decisions

A virtual domain is one that is not stored physically, but rather is computed each time it is required from the domains on which it is computationally dependent. Also, notice that updating a virtual domain is not a legal operation. The virtual nature of the domain is, by definition, not visible to the user.

A domain is a candidate for virtualization if it is computationally dependent on a set of other domains; Ie: p is a candidate for virtualization if $8c(|a|,p)=1. Any of the domains on which it functionally dependent (ie: j where j<|p|) may also be virtual, but there must be a restriction to prevent circular computational dependencies. Namely:

If $\$8c(|r|,P)=\$8c(|y|a)=1$ where $a \in |r|$ then:

$\$8c(|x|.b) \neq 1 \quad \forall b \in |y|, \quad \forall |x|$ where $p \in |x|$

The decision rule is:

For a domain that is currently virtual, If:

(cost(making domain real) + cost(use if domain is real) + cost(maintaining domain if real))

< (cost(using the domain if virtual))

then make it real. Otherwise leave it as virtual.

Note that the cost of maintaining a virtual domain is 0. In the event that a domain is real, then each time any of the domains on which it is computationally dependent is modified, the domain itself must be modified. This is clearly not the case if the domain is virtual.

Similarly, if the domain is currently real, if:

(cost(virtualizing) + cost(use if domain virtual)) <

(cost(use if domain real) + cost(maintaining if real))

then virtualize the domain; otherwise leave it as real.

Separating out the various costs mentioned above, we get:

5.3.1.1 Cost(making domain real)

This involves a serial processing of the relation(s) in which the domains on which it is computationally dependent exist, and computing the value of the virtual domain. It is then appended to the relation, and written out in the database in the new form. Thus there are basically two steps:

locate the domains on which it is computationally dependent, and compute and store the value.

Assuming that the domain in question is domain d, there are two possibilities when \$8c(|p|,d)=1 :

a) \$8d(i,j)=1 ∀j<|p| and \$8x(i)=1 for that i, or

b) \$8d(i,j)≠1 for some j<|p|, and a single i

In case (a), no joins are necessary when retrieving the domains on which d is computationally dependent; they are all in relation i. Computing the value of d consists simply of retrieving a tuple from relation i and computing the value. In case (b), however, there will be at least one join necessary to retrieve all members of |p|, and very possibly several. Case (a) is really a special form of case (b), which is, in fact the general case. If there were some algorithm capable of determining the cost of a serial retrieval for all domains in list |p| for the general case (case(b)) then case (a) would be automatically included. In fact, such an algorithm is also required by the Request Decision Subsystem (RDS) in determining the cheapest way of resloving a request. The concepts involved are identical: how to optimally retrieve all domains required to perform the desired function. This algorithm is thus common to both this situation, and the RDS operations. As such it has been detailed in Chapter IV. For our purposes, it is enough to note that the invocation of the algorithm, given the list of domains |p|, will result in a cost estimate for the <u>cheapest</u> way of performing the request. We will call the cheapest method the 'final' method determined by the algorithm, and the cost of performing it will be the 'cost(final)'.

And so, the computation of cost(making domain real) becomes:

cost(final) + ($5cy(i)r/$4bfe).$4io + $5cy(i)r.$4opc

assuming that the real domain d is inserted in relation i. Ie: the cost is the cost of computing the value of the virtual domain, plus the cost of writing it out in relation i. The component ($5cy(i)r/$4bfe).$4io will become familiar throughout all future decision rules. It takes into account the fact that relations are blocked, and that not each call to retrieve (or insert, update or delete) an entry will necessarily result in a real I/O. The cost component'$5cy(i)r.$4opc' covers the cost of the overhead in each call to XRM. In this way, we take account of the fact that each call involves some expense, but not necessarily an I/O. This will be found in most subsequent decision rules.

5.3.1.2 Cost(use if domain real)

This is basically comprised of the cost of additional storage, plus the cost of retrieval (or other operations) if the domain is real.

cost(additional storage) = 4.$5cy(i).$4sc.$4t

since each domain in XRM is a fullword domain.

At this point, the system would make a decision regarding whether this domain should have an index (see 2 below) - ie: would determine whether $8i(i,j)=1 .

If $8i(i,j)=1 then:

```
cost(use if domain real) = (  $1rd(i,j)**e**.($4io + $4opc)
                           +  ($1rd(i,j)*sn**+$1rd(i,j)*cn*n)  .
                           (($5cy(i)r/$4bfe).$4io            +
                           $5cy(i) r.$4opc)
                           +   (trss/$1rd(i,j)*cn*i(trss))     .
                           ($4io + $4opc)    )
```

If $8i(i,j)=0 then:

```
cost(use if domain real) = (  ((trss/$1rd(i,j)*cn*i(trss)) .
                           ($4io + $4opc)
                           ($1rd(i,j)*s**+$1rd(i,j)*c**n).
                           (($5cy(i)r/$4bfe).$4io          +
                           $5cy(i)r.$4opc)   )
```

Thus, in the event that there is an index, any case where domain j is used as a qualifier in a <qualifier type> of 'e' selection criterion, it is simply a case of using that index. For <qualifier type> of 'n', the index is of no use, and some serial search will be needed. If some other domain in the selection criteria was indexed, then only the resolved set, after using that index, need be serially searched.

If there is no index, then all cases, except those where there is some other domain in the request with an index, a serial serach is required.

5.3.1.3 cost(maintaining domain if real)

The cost of maintaining the domain if it is real is an additional update each time any of the domains on which the new real domain is computationally dependent and in another relation, is updated in any way. This is because if a domain on which j is computationally dependent is in the same relation, then there is no additional I/O, or call to XRM.

For domain j of relation i:

cost = (\$2rd(i,|k|)u**.(\$4io + \$4opc))

∀m<|p| where \$8c(|p|,j)=1 and \$8d(i,m)=0.

5.3.1.4 cost(virtualizing)

There is a choice as to whether the domain is physically deleted from the relation or whether it is just marked as being virtual, and not physically removed. If the domain is not physically removed, then:

cost(virtualizing) = 0.

If it is physically removed, then:

cost(virtualizing) = 2.((\$5cy(i)r/\$4bfe).\$4io
+ \$5cy(i)r.\$4opc)

for i where \$8d(i,j)=1.

Ie: the process of removal involves serially reading and then writing (with the domain removed) each entry in the relation.

The decision whether to physically remove the domain or not is:

If cost(physical deletion) < cost(storage wasted) then physically delete the domain.

cost(physical deletion) is as above.

cost(storage wasted) = 4.$5cy(i) r.$4sc.$4t

Thus the cost(virtualizing) decision is a two-tierred decision rule.

5.3.1.5 cost(use if domain virtual)

(Note: updates of virtual domains are illegal; inserts and deletes are unnecessary. Thus only retrievals and use of the domain as a qualifier are permissible for virtual domains.)

The cost(use if domain virtual) is broken down into two types of use:

- use as a qualifier
- the object of a retrieval request.

5.3.1.5.1 cost(use as a qualifier)

This involves a serial processing and computation of the value of the virtual domain for <u>all</u> entries, and checking that value against the criteria specified in the qualifier. The 'cost(final)' is the same as that described in 1.1.1 above.

For cases where there was some other domain in the selection criteria that was indexed, the size of the set to be serially searched is (on average) (trss/$1rd(i,j)*c**i(trss)).

cost(use as a qualifier) = (cost(final).($1rd(i,j)*s** +
$1rd(i,j)*c**n)
+
$1rd(i,j)*c**i(trss).cost(final')
)

where cost(final') is the same as cost(final), <u>except</u> that all instances of $5cy(i)r in the algorithm are replaced by 'trss'.

5.3.1.5.2 cost(retrieval)

cost(retrieval) = ($2rd(i,j)r**.cost(final))

This ends the discussion of virtualizing decisions. We move on now to encoding decisions.

5.3.2 Encoding Decisions

Encoding decisions are made only for domains which have a data type of 'character';
Ie: where $8_(j)cu=1

For purposes of this thesis, we will consider only one coding scheme. This was done simply to avoid becoming to voluminous, as the number of possible coding schemes is potentially infinite. Furthermore, the purpose here is not to be complete, but rather to present an approach.
The scheme that will be employed here is the encoding of character strings as bit strings of length !(log($6(j)q)/log 2). ('!' means here the next highest integer, unless the expression evaluates to an integer, in which case that is the value used.) This may only be done if the number of unique values in the domain is constant (ie: $6(j)q is constant)
The encoding decision becomes:

If the domain (say d) is not currently encoded, then encode it if:

(cost(encoding) + cost(use if encoded)) <

(cost(extra storage) + cost(use if unencoded))

Similarly, if it is currently encoded, then decode if:

(cost(decoding) + cost(extra storage) + cost(use if decoded))

< (cost(use if encoded))

Breaking down these costs into the individual components:

5.3.2.1 cost(encoding)

The cost of encoding domain d consists of the serial processing of all relations in which domain d appears, and replacing the id of the character string with a bit string of the required length. Additionally, there is the cost of building, and maintaining the encoding relation.

cost(encoding) = (2.SUM(($5cy(i)r/$4bfe).$4io + $5cy(i)r.$4opc)

+ (($6(d)q/$4bfe).$4io + $6(d)q.$4opc))

∀i such that $8d(i,d)=1 ie: all relations in which d appears.

5.3.2.2 cost(decoding)

The cost of decoding a domain and storing the actual values rather than the encoded values is identical to that of encoding.

Ie: cost(decoding) = cost(encoding) see 1.2.1

5.3.2.3 cost(use if encoded)

The way an encoded domain is used is to employ the code value as a primary key for the encoding relation. This means that each time the domain is the object of a retrieval or an update, or each time it is used as a qualifier, there will be an additional I/O and an additional call to XRM to retrieve either the code or the value (depending on whether it is being used as a qualifier or is the object). Thus:

cost(use if encoded) = 2.($1rd(i,d)***** + $2rd(i,d)|r,u|**)

. ($4io + $4opc)

5.3.2.4 cost(use if unencoded)

Since use of the domain if encoded involves an additional I/O and call to XRM each time the domain is used, it follows that the use of the domain if unencoded should be 1/2 of that if encoded; the additional retrieval is avoided. Thus:

cost(use if unencoded) = ($1rd(i,d)***** + $2rd(i,d)|r,u|**) . ($4io + $4opc)

5.3.2.5 cost(extra storage)

The cost of the additional storage required to store the unencoded values will be the difference between the storage required if the domain is unencoded, and that required if the domain is encoded.
The cost of storage if unencoded will be a fullword for each entry in each relation in which the domain appears. Ie:

cost(storage if unencoded) = 4.$4sc.$4t.SUM($5cy(i)r) ∀i where $8d(i,d)=1

The cost of storage if the domain is encoded will be:

- the overhead for the extra relation - about 100 bytes in XRM
- two fullwords per entry in the encoding relation; the first being the code, and the second the actual value, and
- the sum of all the space in each relation in which the domain appears.

Ie:

cost(storage if encoded) = ((SUM(!(log $6(d)q/log 2) + 8.$6(d)q + 100) . ($4sc.$4t)

∀i such that $8d(i,d)=1.

The additional storage is thus the difference between these two. Note that the additional storage may be negative in the event that there is only a small set of values, given the overhead. This would not alter the decision rule in any way, as it would way in favor of not encoding, which is what should happen.

Thus: cost(extra storage) =

(4.SUM($5cy(i)r

- (SUM(!(log $6(d)q/log 2) + 8.$6(d)q + 100))

.$4sc.$4t

This concludes the decision rules regarding encoding of domains. We proceed now to indexing decisions.

5.4 Indexing Decisions.

In XRM, as described in Chapter II, any domain in the system may have an inversion, or index, created for it. When there is an inversion on some domain, and that domain is used as a qualifier with <qualifier type> 'e', then retrievals, or locating of tuples with the specified value in that domain is extremely rapid. There are some schemes that address themselves to indexes for qualifiers when the <qualifier type> is 'n', but we shall not address such schemes here. For our purposes, we are interested in an approach, and the approach taken here may be easily extended to include qualifiers of type 'n'.

In XRM indexes are built in a specific way. Specifically, an index entry is a 'value/id' pair, where the value is the primary key. Indexes are implemented as binary relations. Given a value, it is used as the primary key to locate the id of a tuple containing that value in that domain. However, the use of a value as a primary key has severe limitations. There is no reason why a value should not appear in many entries, and in fact, that is usually the case (except in the case of primary keys). There is thus some method needed which allows for this factor.

The method employed in XRM is to chain together all id's of entries that have any one value in the specified domain. Thus, while there would ordinarily be two fullwords (8 bytes) for each index entry, consisting of a 'value/id' pair, we now have each index entry consisting of a 'id/pointer' pair, with the start of the chain having the 'id' replaced by a 'value'. There is therefor, one additional word of storage required per chain over the strict binary relation implementation. Furthermore, while many schemes have several levels of indexing, such as that found in ISAM, the XRM index is only one level deep. The decision rules, however allow for a multi-level index.

The decision rule for indexing is:

If (cost(storage) + cost(projected use <u>with</u> index) + cost(building index))

< cost(projected use without index)

then build an index.

Similarly, if an index already exists for a domain, then eliminate the 'cost(building index)' part of the decision rule.

The projected use of the domain is based on that experienced in the preceding time period: $4t. There is an implicit assumption in all these decision rules that use will continue unchanged, which is, in fact, a reasonable assumption to make. In the event that use changes, the SDS will again be invoked, and will proceed under the same assumption.

We proceed now to break down the components of the decision rule.

5.4.1 cost(storage)

cost(storage) = SUM(((# entries at level i) . (space per entry at level i)) + (overhead for level i)), i=1,...L where L is the number of levels of index.

As stated above, in XRM, L=1. The following is also true of XRM:

- overhead per inversion is approximately 50 bytes
- space per entry is 8 bytes, plus 4 bytes per chain (see above)

Thus, for XRM:

cost(storage) = (50 + 8.$5cy(i)r + 4.$6(d)q).$4sc.$4t

for an index on domain d of relation i.

5.4.2 cost(projected use with index)

This component of the decision rule can be further broken down into three sub components. These are:

- cost(retrievals)

. cost(decoding index)

. cost(maintaining index)

5.4.2.1 cost(retrievals)

If there is an index on domain j of relation i, then any time that domain j is used as a qualifier of type 'e', the index can be employed to limit the size of the resolved set of entries. In the event that the qualifier type is 'n', then the index is of no value, and a serial search is required. Since this is the case throughout all of these decision rules, we can eliminate those cases where the qualifier type is not 'e'. The decision rules specified here will thus include only <qualifier type> 'e' decision variables.

We assume, furthermore, that entries retrieved are distributed randomly throughout the relation.

The subcomponent cost(retrieval) thus becomes:

cost(retrieval) =

(($5cy(i)r/$6(j)q).($1rd(i,j)*se* + $1rd(i,j)*ce*n)

+ MIN((($5cy(i)r/$6(j)q).$1rd(i,j)*ce*i(trss)),

(trss/$1rd(i,j)*ce*i(trss)) . ($4io+$4opc)

If a tentative index decision has been made for some other domain in relation i, say domain j', at the time at which domain j is being evaluated to see whether it warrants an index, then some requests that were previously compound requests in which no other domain had an index will now become requests in which some other domain in the selection criteria has and index. It is therefor necessary to transfer some of the requests from decision variable $1rd(i,j)*ce*n to $1rd(i,j)*ce*i. If there has been a tentative decision to drop an index on some domain in the relation, then transfer the requests in the oposite direction.

The number of requests transferred is a function of the number of compound requests that a given domain was involved in as a fraction of all compound requests. Ie: Transfer the following number of requests from $1rd(i,j)*ce*n to $1rd(i,j)*ce*i:

$$\frac{\$1rd(i,j')*ce**}{\$1rd(i,*)**e**} \cdot \frac{\$1rd(i,j)*ce**}{\$1rd(i,*)**e**} \cdot \$1rd(i,j)*ce**$$

5.4.2.2 cost(decoding index)

This is a function of both the CPU overhead time involved in the decoding of an index, as well as the necessary number of I/O's to get the index into primary memory. However, since

CPU time is so small in comparison with I/O time, the decision rules presented here will not take into account CPU overhead in decoding indexes.

Note that the maximum index blocking factor ($4bfx) is (($4p/2) - $6(d)q) .

The cost of decoding the index is thus the number of I/O's necessary to bring the index into primary memory. Ie:

cost(decoding index) =

($1rd(i,j)**e** . ($5cy(i)r/$4bfx) . $4io)

5.4.2.3 cost(maintaining index)

The cost of maintaining the index is a function of the number of new entries that are made in the relation, as well as of the number of times a value in the domain is updated. What is assumed for the purposes of the decision rules presented here is that the insertions and updates all require the index to be brought into primary memory. However, to be strictly correct, the decision rules should be concerned with the length of a series of inserts or updates involving the domain in order to take into account the fact that the index need not be brought into primary memory separately for each operation.

cost(maintaining index) =

(\$2rr(i)ien + \$2rd(i,j) |u,d|**).(\$4io + \$4opc)

5.4.3 cost(building index)

The cost of building the index will be the cost of a serial retrieval of each entry in the relation, and a write operation to the index. Notice that the rule below has only the overhead of a single call to XRM. This is because inversion is accomplished by a specific XRM routine.

cost(building index) =

\$4opc+\$4io.((\$5cy(i)r/\$4bfe)+(\$5cy(i)r/\$4bfx))

5.4.4 cost(projected use without index)

In the event that there is no index, any time the domain is used as a qualifier in a simple query, or a compound quiry in which no other domains in the qualifier had an index, a serial retrieval is necessary.

```
cost(projected use without index) =
        (($5cy(i)r/$4bfe).$4io)   +
        $5cy(i)r.$4opc).($1rd(i,j)*se*+$1rd(i,j)*ce*n))
        +  MIN(  (($5cy(i)r/$4bfe).$4io   + $5cy(i)r.$4opc)   .
        ($1rd(i,j)*ce*i(trss))  ,
              ((trss/$1rd(i,j)*ce*i(trss)).($4io+$4opc)  )   )
```

This concludes the decision rules for indexing decisions. We proceed with decision rules for factoring.

5.5 Factoring Decisions.

Factoring decisions are decisions regarding the storing of aggregations (or factored data) as opposed to computing them each time they are required. The aggregations which this system recognizes are:

- MAX
- MIN
- COUNT
- UNIQUE   the number of unique values
- AVERAGE
- SUM

This information applies only to numeric domains.

In addition, the following should be noted:

- storage for these aggregations are always reserved in the system tables, and so the cost of storage is not considered in the decision rules.
- COUNT is always maintained by the system, as the RDS uses it continuously.
- UNIQUE is no more than the COUNT of the underlying domain, and so is always maintained

If SUM is stored, there is no need to store AVERAGE. The reverse is not true, howver because of possible roundoff errors.

The factoring decision is:

If cost(maintaining aggregation) <
cost(computing aggregation)
then store it. If not, compute it each time.

5.5.1 cost(computing aggregate)

This consists of a linear processing of the relation, and so the cost is simply:

```
(($5cy(i)r/$4bfe).$4io  +
        $5cy(i)r.$4opc ).( $2rd(i,j)r(<aggr>)**
where <aggr>::= SUM | AVERAGE | MIN | MAX
```

5.5.2 cost(maintaining aggregation)

This involves computing the aggregation once, and then maintaining it, or updating it each time a value in that domain is updated, deleted or inserted.

```
cost(maintaining aggregation) =
          (($5cy(i)r/$4bfe).$4io + $5cy(i)r.$4opc)
          +   ($2rd(i,j)|u,d|**  +  $2rr(i)ien)  .  ($4io  +
          $4opc)
```

This concludes the discussion of factoring decisions. We proceed now with permanent join decisions.

5.6 Permanent Join Decisions

This decision rule is employed in the event that some new domain(s) has (have) been added to the system and have been set up in separate relations to avoid restructuring the existing relation. The format of the new relation will be the primary key of the existing relation in which the new domain(s) belong(s), and the new domain(s). This means that any time these new domains are used in conjunction with any of those in the existing relation, a join is required, or more precisely, another retrieval is required. This is because both relations have the same primary key, and so a join is a trivial matter.

If the relations are left separate, then there will be additional retrievals required in satisfying certain requests. The reverse is not true. That is, if the relations were to be permanently joined (restructuring were to occur), there is no case where the fact that they are joined permanently would result in additional retrievals over the case where they were left separate. This being so, we are able to drop the concept of cost(projected use) and concentrate rather on the cost(projected use <u>premium</u>).

sp;The decision then becomes:

If (cost(restructuring) + cost(storage if restructured))

< (cost(projected use premium) + cost(storage if not restructured))

then restructure the relations into a single (permanently joined) relation. Breaking down the cost components, we have:

5.6.1 cost(projected use premium)

This is a case of several additional retrievals being necessary whenever any domain(s) in the new un-joined relation are used in a request together with some of those domains in the existing relation. Ie:

`cost(projected use premium) =`

`$3rd(i,|p|)d(j,|p|) . ($4io + $4opc) where:`

`p<|p| ∀p such that $8p(i,p)=1`

This statistic is maintained separately by the RDS - ie: the number of times each relation is joined to each other relation.

5.6.2 cost(storage if restructured)

The cost of storage if the relations are restructured will be 4 bytes for each entry for each domain in the new relation excluding the primary key domains. Ie:

`cost(storage if restr)= 4.$5cy(i)r.$4sc.$4t.SUM($8d(i,j))`

∀j such that $8p(i,j)=0

5.6.3 cost(storage if not restructured)

This is similar to the computation of 5.6.2, except that the restriction that the domain not be in the primary key is lifted. Ie:

cost(storage if not restr) =
4.$5cy(i)r.$4sc.$4t.SUM($8d(i,j))
where i is the new relation

5.6.4 cost(restructuring)

The restructuring of two relations with the same primary keys consists of a serial processing of one relation, using its primary key to retrieve from the other relation, and writing the new joined entry out again. In other words, three operations for each entry. If i is the existing relation, and j is the new relation, then:

cost(restructuring)= 3.($5cy(i)r/$4bfe).$4io
+ 3.$5cy(i)r.$4opc

This completes the decision rules for restructuring decisions. We proceed now with decision rules responsible for establishing derived relations.

5.7 Derived Relation Decisions.

It is important to point out that derived relations are created solely for reasons of efficiency, and so decisions to create derived relations are quite independent of structuring decisions of the type mentioned above, and the third normalizing process.
The choice exists basically between storing a virtual relation defined by some user for some specific application (or set of applications) or simulating that virtual relation each time it is referenced. If:

cost(simulating virtual relation) <
 (cost(storage for derived relation) + cost(use of derived relation) + cost(creating derived relation) + cost(update overhead)

then continue simulating the virtual relation. If not, then it is cheaper to store it.
Similarly, if a derived relation is stored, the decision to cease storing it and to return to simulating the virtual relation would be the same as that above, except that it would exclude the 'cost(creating derived relation)'.
All references to 'cost(final)' are the same as those made earlier in the chapter.

5.7.1 cost(simulating virtual relation)

A virtual relation is simulated by performing joins in the user workspace of various real relations that are required for a particular request.

cost(simulating virtual relation) =

cost(final).($1vr(m)*nu*
+ 1/$6(j)q.($1vd(m,*)*se* + $1vd(m,*)*ce**)
+ 1/2($1vd(m,*)*sn* + $1vd(m,*)*cn**))

5.7.2 cost(storage for derived relation)

The storage for a derived relation will consist of the overhead per relation, and the storage for each of the domains of the derived relation. The cardinality will be approximated by the maximum cardinality.

cost(storage for derived relation) =

(100 + $5cy(m)d.$5#d(m)v.4) . $4sc.$4t

where the derived relation is relation m, and $5cy(m)d = $5cy(¢)r and ¢=i x i' x... ∀i such that $8d(i,j)=1 and ∀j where $8d(m,j)=1.

5.7.3 cost(use of derived relation)

Before determining the use cost of the derived relation, the SDS would make an inlexing decision for the domains of the virtual relation (see 2 above except, substitute 'v' for all 'r' in relation types). A derived relation may then be treated in an analogous manner to a real relation. Ie:

cost(use of derived relation m) =

($5cy(m)d/$4bfe).$1vr(m)*nu*.$4io

+ $5cy(m)d.$1vr(m)*nu*.$4opc

+ SUM(($8i(m,j).($5cy(m)d/$6(j)q). $1vd(m,j)*|s,c|e**

. ($4io + $4opc)

+ (1-$8i(m,j))($5cy(m)d/$4bfe). $1vd(m,j)*|s,c|e* . $4io

+$5cy(m)d.$1vd(m,j)*|s,c|e* . $4opc)

+ (($5cy(m)d/$4bfe).$1vd(m,j)*|s,c|n** . $4io

+ $5cy(m)d.$1vd(m,j)*|s,c|n** . $4opc)

)

∀j where $8d(m,j)=1 .

5.7.4 cost(creating derived relation)

The cost of creating the derived relation is no more than the cost(final), since that is, in fact the optimal way of creating it.

5.7.5 cost(update overhead)

The update overhead for a derived relation involves an additional I/O and call to XRM for each change to any of the domains in the real relation that also appear in the derived relation, m. Ie:

cost(update overhead) =

SUM($2rd(i,j)|u,s,i|** . ($4io + $4opc)) ∀j where $8d(m,j)=1, and ∀i where $8d(i,j)=1.

This completes the discussion of the SDS decision rules. It can be seen that these rules are quite modular in that any major decision - for example, Derived Relation Decisions - are composed of several subdecisions. Any or all of these subdecisions can be replaced without affecting any other part of the SDS.

These rules will be applied in the scenario of Chapter VII.

The Request Decision Subsystem - (RDS)

The RDS is responsible for overseeing any requests that are made against the database. More specifically, it is responsible for the following functions:

- determine whether the request is legal - ie: check access control information and decide based on that whether to perform the request.
- determine whether the request is feasible - ie: determine whether it can logically be satisfied, or whether the system requires more information to resolve the request.
- determine the most efficient way of satisfying the particular request, assuming it is deemed 'feasible'.
- update the relevent decision variables.

For purposes of this thesis, we will omit the question of the legality if the request. It is envisioned that the access control mechanisms will be implemented at the real relation level. The creation of virtual relations can be controlled in such a way as to make the data that the user sees in a virtual relation only that which he (it) is permitted to see. Any data the user is not authorized to see will be removed from the data during the mapping process, thus making the fact that there is some data not being supplied invisible to the user. Thus, security can be implemented as restrictions of real relations.

We proceed now with the collection of algorithms that the RDS will contain for the resolving of requests. We shall not detail the points at which decision variable updates are performed for reasons of making an already difficult section more unreadable. Instead, it should be fairly clear at which point specific decision variables will be updated from the context of the discussion at that point. Finally, note that decision variable updates are not actually made until the completion of Step 5. All updates until that point are <u>tentative</u> and only become final at the conclusion of the step. The reason for this approach will become clear from the iterative nature of the RDS.

Note that the notation developed thus far will be continued here. In addition, the decision variable '$5cy(i)e' should be taken to read '$5cy(i)r <u>OR</u> $5cy(i)d'.

Step 1.

The purpose of Step 1 is to establish a set of truth functions regarding the domains appearing in the request. The truth functions set up are:

- $8od(k) = 1 for all k that are object domains in the request. If an entry is specified (ie: all domains in a relation) then such a truth function is set up for every domain in the relation.
- $8eq(k)=1 for all domains k that appear as qualifiers with <qualifier type> 'e'.
- $8nq(k)=1 for all domains k that appear in the request as qualifiers with <qualifier type> not 'e'.

Note that k may be a role name instead of a domain name.

Step 2.

If the list of domains appearing in the request is |R|, then for each k<|R| find all relations i such that:

$8r(i,k)=1 or $8d(i,k)=1, and

$8x(i)r=1 or $8x(i)d=1.

but if $8x(i)d=1, i should not be a derived relation whose derivation involved a restriction.

This step finds all relations in which each of the domains in |R| appears. This step produces a list of relations for each domain k ie: |i(k)|. These lists are all members of a single list: ||i(k)||, where L(||i(k)|| = L(|R|).

If for any |i(k)| we have L(|i(k)|)=0 then the request is 'infeasible', because some role- or domain name is used in the request which has not been defined. It is also possible that a domain appears more than once in a single relation, and if that is the case the request must supply role names. Ie:

$8d(i,k)=0, and $8r(i,k')=1 and $8r(i,k'')=1 where k' and k'' are role names.

Step 3.

This step simply establishes a list of relations which will eventually (at the end of the alogorithm) become the optimum list of relations to satisfy the request.

Ie: set up list |final|, initially 'infeasible', and cost(|final|)=2**30 (or some maximum number).

Step 4.

This step finds all possible combinations of relations that can satisfy the request. The results of the step is a list of relations that contain all the domains necessary to satisfy the request, based on the set of truth functions established in Step 1. The procedure is as follows:

$\forall |i(k)| < ||i(k)||$:

  Initialize $|x|=|i(k)|$

  $\forall j \neq k$ if $|x| \cap |i(j)| = \emptyset$ then:

    $|x|=|x| \cup |i(j)|$

  Order list |x| in ascending order, and see if that list already has been 'done'. If yes, go on to the next |i(k)|. If not then save a copy of this |x| as 'done' and do Step 5.

  After completing Step 5:

    If $cost(|x|) < cost(|final|)$ and |x| is not 'infeasible' then set:

    |final|=s', and cost(|final|)=cost(|x|) where s' is the collection of set theoretic operators generated by Step 5.

The results of this step (after repeated invocations of Step 5) is the optimal list |final| as a collection of set theoretic operators, s', and an associated cost, cost(|final|).

Step 5.

This step consists of a series of substeps whose function it is to determine the lowest cost(|x|) - or more precisely, the lowest cost for resolving the request via relations in |x| - for a given |x| from Step 4.

If any relation i<|x| can not be joined to any other relation i'<|x| then then |x| is 'infeasible', and the evaluation stops.

We proceed now with a detailed algorithm for determining the minimum cost(|x|).

Step 5.1

This step finds all relations in |x| with the same primary key, and the cheapest way of ordering the necessary joins, and restrictions.

Repeat ∀i<|x| not already 'done':

Set |x'| to <null>, and cost(|x'|)=2**30 (or some large number).

Now find all relations j (say |j|) in |x| with the same primary key:

Ie: Find all j such that $8p(j,k)=1 ∀k where $8p(i,k)=1.

This results in a set |j'|=i U |j|

∀a<|j'| do the following:

Set |r'| to <null>.

For each k∈a where $8eq(k)=1 and $8i(a,k)=1, set r=($5cy(a)e/$6(k)q). Insert this r in |r'| such that |r'| is in ascending order.

If $8eq(k)=1 ∀k where $8p(a,k)=1 then r=1, and insert in ascending position in |r'|.

This case is where all domains of the primary key are qualifier domains with <qualifier type> 'e'.

Also, if any qualifier domain with <qualifier type> 'e' is unique, then r=1.

Ie: If $8eq(k)=1, $8u(k)=1 and $8i(a,k)=1 then r=1, and insert r in ascending order in |r'|.

Choose w = first member of |r'|; ie: the smallest r∈|r'|.

Then cost(|j'|) = w.($4io + $4opc).L(|j'|)

If cost(|j'|) < cost(|x'|) then:

cost(|x'|)=cost(|j'|), and

|x'|=|r'|

Reset |r'| to <null>.

The case now is where r will resolve, but there is no index on any of the qualifier domains:

If $8eq(k)=1 and $8u(k)=1 then r=1, and insert in ascending order position in |r'|.

For each k∈a where $8eq(k)=1 and $8d(a,k)=1, but $8i(a,k)=0, set:

r=($5cy(a)e/$6(k)q), and insert r in ascending order in

|r'|.

(Note that the reason that r=1 when $8u(k)=1 is that $6(k)q=$5cy(a)e.)

After completing all k<a, pick w = first member of |r'|; ie: the smallest r.

Then cost(|j'|) = ($5cy(a)e/$4bfe).$4io

+ $5cy(a)e.$4opc

+ (L(|j'|)-1).w.($4io + $4opc)

If cost(|j'|)<cost(|x'|) then:

cost(|x'|) = cost(|j'|), and

|x'| = |r'|

Reset |r'| to <null>.

Now, for each k<a where $8nq(k)=1 and $8d(a,k)=1, set r=1/2.$5cy(a)e, and insert r in |r'| in ascending order.

After all k<a are completed, choose w=first member of |r'|; ie: for the smallest r. Then:

cost(|j'|) = ($5cy(a)e/$4bfe).$4io

+ $5cy(a)e.$4opc

+ (L(|j'|)-1).w.($4io + $4opc)

If cost(|j'|)<cost(|x'|) then:

cost(|x'|) = cost(|j'|) and

|x'| = |r'|

If at this stage |x'| is null, then a serial retrieval is required, since there were no qualifiers in the request. Proceed as follows:

Find a<|j'| such that $5cy(a)e is a minimum ∀a<|j'|.

Mark a as 'done', and set s=a.

For each k≠a, k<|j'|, do the following:

Mark k as 'done'.

If ∀b where:

($8eq(b)=1, $8nq(b)=1 <u>or</u> $8od(b)=1) and

($8d(s,b)=1 <u>and</u> $8d(k,b)=1) then no join is needed, since all domains needed from k are already in s.

Otherwise, the set theoretic operators become:

s=s(|p|) * k(|=,p|) ∀p<|p| where $8p(a,p)=1, and

cost(|x'|) = ($5cy(a)e/$4bfe).$4io
+ $5cy(a)e.$4opc
+ $5cy(a)e.($4io + $4opc).(L(|j'|)-1)

If, however, |x'| was not <null> at the end of the above procedure, proceed as follows:

Choose a=first member of |x'| (ie: where r is smallest), and mark a as 'done'.

Set s=a.

The set theoretic operators then become:

s=s(|p|) R (|θ,p|) ∀p<|p| such that \$8eq(p)=1 <u>OR</u> \$8nq(p)=1,

where 'θ' is the qualifier on domain p. (see Appendix 2, Page 175)

Now, for all k≠a, k<|j'| do the following:

Mark k as 'done'.

If ∀b where:

(\$8eq(b)=1, \$8nq(b)=1 <u>or</u> \$8od(b)=1) and

(\$8d(s,b)=1 <u>and</u> \$8d(k,b)=1) no join is necessary, since all domains needed from relation k are already in s.

Otherwise, establish the following set theoretic operators:

If \$8eq(b)=0 and \$8nq(b)=0 ∀b<k, then

s=s(|p|) * k(|=,p|) ∀p<|p| where \$8p(a,p)=1.

Or, if \$8eq(b)=1 or \$8nq(b)=1 for some b<k, and (\$8d(s,b)=0 <u>and</u> \$81(k,b)=1) then set theoretic operators become:

s=(s(|p|) * k(|=,p|)) R (|θ,t|) ∀t<|k| such that \$8eq(t)=1 or \$8nq(t)=1, and θ is the qualifier type on domain t (see Appendix 2, Page &pno3).

Once all k<|j'| have been so included in the set theoretic operators, they can be removed from the list of relations to be joined (ie: |x|) and replace them all by the single relation that would result from s. Also, set up truth functions as follows:

$8d(s,b)=1 ∀b where $8d(k,b)=1 for k<|j'|, and

$8p(s,b)=1 ∀b where $8p(a,b)=1.

Also set $5cy(s)v = r.

Note that r in this case is, strictly speaking, an upper limit on the cardinality of s, since other qualifiers may well result in reducing the size of r.

At the completion of Step 5.1, there exists a collection of (virtual) relations |s'| each generated as outlined in this step. Each s<|s'| is a relation consisting of all real (and derived) relations in |x| that have the same primary key, and is restricted as required by the qualifiers in the request.

If L(|s'|)=1 then steps 5.2 , 5.3 and 5.4 may be omitted. The algorithm continues in this case with where it left off in Step 4.

Step 5.2

This step is responsible for joining in the most efficient manner, all those relations s<|s'|, in the case where s<|s'| contains the primary key of (but does not have the same primary key as) t<|s'|, t≠s.

Ie: $8d(s,k)=1 ∀k where $8p(t,k)=1, s,t<|s'|.

If there is no s and t where this is the case, proceed to Step 5.3. Otherwise continue with the algorithm of Step 5.2.

Assuming s contains the primary key domains of t:

check to see whether s already contains all the needed domains (for this request) in t, and if so, remove t from |s'| and continue with some other t<|s'|.

If $8od(k)=o ∀k where $8d(t,k)=1 and $5cy(t)v=1 then establish 2 set theoretic operators:

1) t - ie: leave t as it is in |s'|, and

2) s'=s(|p|) R (|=,p|) p<|p| ∀p where $8p(t,p)=1, and values for the domains are obtained from (1).

cost(|s'|)=cost(t) + cost(s).

This means that if there are no domains that are to be

retrieved from t, and t will resolve to a single entry ($5cy(t)v=1), then resolve t and use the values for the primary key domains as additional qualifiers in s.
If $8od(k)=1 for some k where $8d(t,k)=1 then proceed as follows:

s'=(s(|g|) * t(|=,g|)) R (|θ,p|)
where g∈|g| ∀g where $8p(t,g)=1, and
∀p∈|t| such that $8eq(p)=1 <u>or</u> $8nq(p)=1, and
θ is the qualifier type on domain p (see Appendix 2, Page &pno3).

cost(s')=cost(s) + 2.$5cy(s)v.($4io + $4opc)

Remove t from the list of relations to be joined (ie: |x'|), and set up additional truth functions for s' as follows:

$8d(s,b)=1 ∀b where $8d(t,b)=1.

At the conclusion of this step, there exists a collection of virtual relations |s''| each generated from some member(s) s in |s'|. Furthermore, these relations are <u>only</u> joinable in a particular way - namely, as in Step 5.3.

Step 5.3

This step is responsible for joining relations s'<|s''| from step 5.2 (or 5.1) to yield a single relation containing all object domains in the request. Note that by this stage, all qualifier domains will have been employed in the necessary restrictions.

We proceed as follows:

Choose t<|s''| such that $5cy(t)v is the minimum of all relations in |s''|.

Then, ∀a<|s''|, a≠s do the following:

If $8d(t,k)=1 ∀k where :

($8eq(k)=1, $8nq(k)=1 or $8od(k)=1), and $8d(a,k)=1, then t contains all the domains neccessary for the request that are in a. So, remove a from |s''| and continue with the next a<|s''|.

See if $8d(t,k)=1 and $8d(a,k)=1 for some k. Ie: see if any two relations in |s''| contain a common domain. If not, try some other a<|s''|.

If so:

t=t(k) * a(k), and

cost(t)=cost(t) + cost(a) + ($4io + $4opc).$5cy(t)v.$5cy(a)v

Remove relation a from |s''| and add the necessary set of truth functions. ie: $8d(t,k)=1 ∀k where $8d(a,k)=1.

At this stage, if L(|s''|)=1, then we have found the cheapest method for joining all relations in |x|, and we continue where we left off in Step 4.

If this is not the case, and L(|s''|)>1, then the request is 'infeasible' with only the relations in |x|, and some other relations must be found that will allow the request to be logically satisfied. This is the function of Step 5.4.

Step 5.4

This step attempts to find relations which, although not specified in |x| will allow the request to be completed. Finding such 'intermediary' relations can be accomplished as follows:

5.4.1) Set up list |b'|=<null>

5.4.2) Choose some a<|s''|

5.4.3) Find some relation b such that b∉|x| and:

$8d(a,j)=1 and $8d(b,j)=1.

5.4.4) If found, set |b'| = |b'| U b

5.4.5) If not found, then the request is 'infeasible', and continue where we left off in Step 4.

5.4.6) See if $8d(b,k)=1 and $8d(t,k)=1 for some k, t<|s''|, t≠a, b<|b'|

5.4.7) If yes: then set |x|=|x| U |b'|, and remove relation t from |s''|, replacing it with the single relation:

a= (a(j) * b(j))(k) * t(k).

Continue with Step 5.4.12 .

5.4.8) If not, try 5.4.6 for some other t<|s''|, t≠a

5.4.9) If that fails, find relation c∉|x|, c≠b such that:

$8d(b,j)=1 and $8d(c,j)=1 for some j.

5.4.10) If found, |b'|=|b'| U c, and repeat from 5.4.6 .

5.4.11) If not, the request is 'infeasible', and continue from where left off in Step 4.

5.4.12) If $L(|s''|) \neq 1$, repeat 5.4.1 through 5.4.11 .

5.4.13) If $L(|s''|) = 1$, then restart step 5 again from 5.1 with the new $|x|$.

This completes the alogorithm for optimally satisfying requests. Very briefly, it works as follows:
First find all relations in the list that have the same primary keys. See which of those will resolve to the smallest set of data, and do a join of that relation with others of the same primary key.
After relations with the same primary keys have been joined, see if any one relation contains the primary key of any other relation. If he retrieved primary key values to retrieve from the other relation.
After joining all relations by primary keys that can be joined in that way, join remaining relations on some common domain.
If there is no common domain, find some other relation that has domains common to both.

This, then, completes the discussion of the RDS. As stated earlier, the RDS is also responsible for updating decision variables, and the appropriate points for performing this function can be surmised from the description of the algorithm.

We proceed now to a scenario in which the decision rules developed in Chapters V and VI will be applied.

Scenario for application of Decision Rules.

This chapter presents a brief scenario that applies some of the decision rules developed in Chapters V and VI. Not all of these rules be used in the scenario, but a representative number of them will be, and that will serve to illustrate the use of others.

Consider a company divided into Departments, each with its own Manager. Each Department employs Employees, who are assigned to work on one Project at a time, and all projects fall within a single Department. Each project requires certain Parts, which are provided by the company's Suppliers.

If we were to attempt to establish a company data base for this company we would have to:

- specify the entities in which we are interested,
- determine what data we want to maintain about each of these entities, and
- determine how these entities interact.

The interactions can perhaps best be done diagramatically. Given the company structure above, we might diagram the interaction between the entities as it appears in Figure 7.1. The entity at the head of an arrow is, in some sense 'owned' by the entity at the tail. *

But this diagram is not sufficient to express certain aspects of the structure.

For example, for any one Manager, there is only one Department while any Manager may have several Projects under his control. We will introduce an '=' near the head of the arrow to signify the one-to-one nature of the relationship. In the terminology of the truth functions of Chapter IV, this is a mutual dependency.
Ie: given one entity, the other entity is uniquely determined, and vice versa. This is shown in Figure 7.2.

One other case is not expressed in Figure 7.2; namely the difference between cases where one entity is uniquely determined by another, and cases where it is not.

----

* The concept of ownership is the same as that found in the network model. See (7)

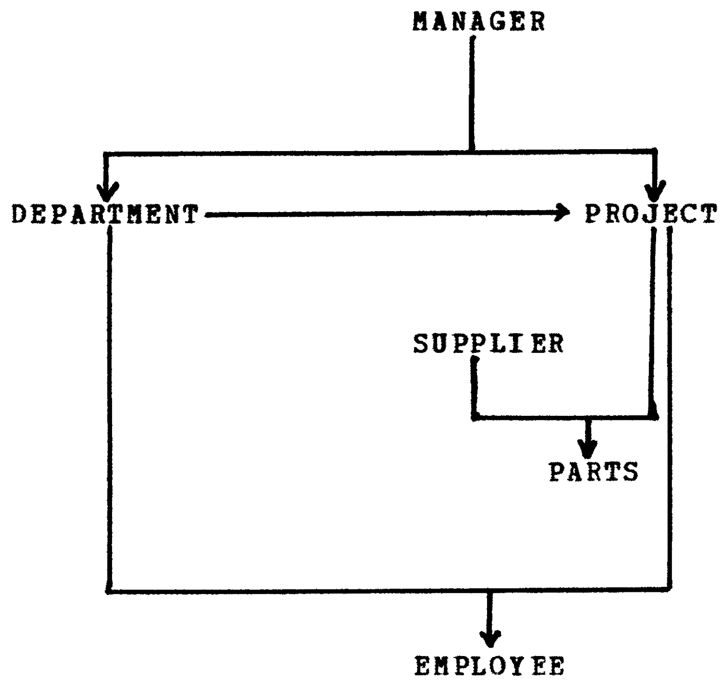

Figure 7.1

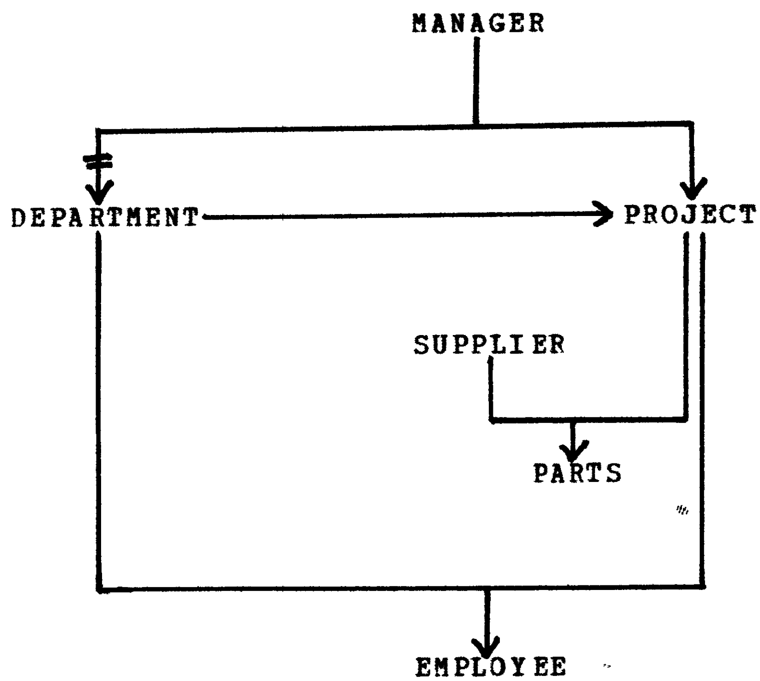

Figure 7.2

The former is a case of a functional dependency ** or a one-to-many mapping. The latter is a many-to-many mapping. An example of the latter is Supplier and Parts, where several Suppliers may supply the same Part, and one Supplier might supply many Parts. A possible method for diagramatically distinguishing between these two cases would be a double-headed arrow for many-to-many mappings. Figure 7.2 is updated to include this concept in Figure 7.3. Now that we have a clear concept of the relationships between the entities, we can begin to consider what information, or attributes, we wish to keep about each entity. ***

Assume for purposes of the scenario that the attributes to be maintained for each entity are as specified in Figure 7.4.

Now notice that some attribute (or combination of

** Functional dependency, as explained in Chapter II, means simply, given one entity, the other is uniquely determined, but the reverse is not true. For example, given an Employee, his Department is uniquely determined since an Employee can only belong to one Department.

*** Note that consideration of attributes and consideration of interrelationships between entities are orthoganal, and as such, may be done in any order. The order presented here is by no means mandatory.

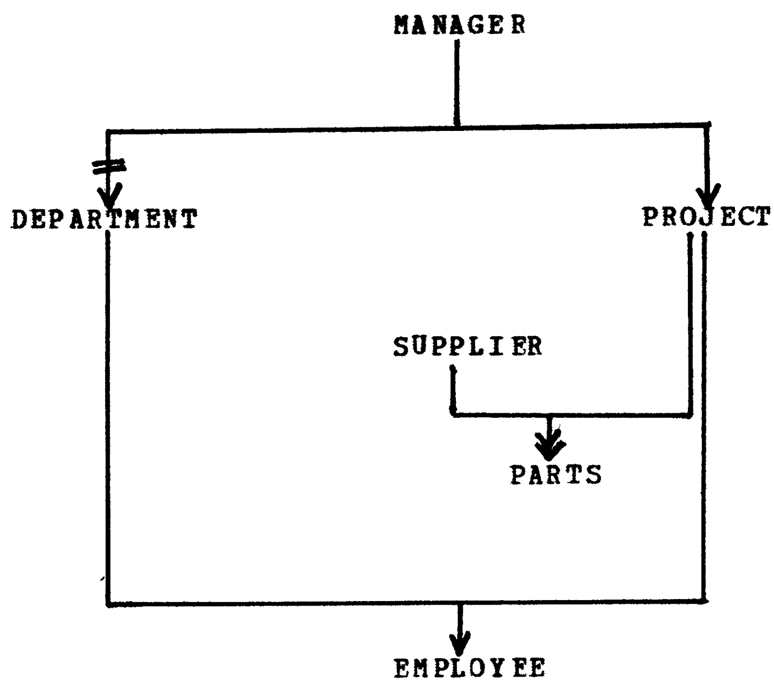

Figure 7.3

Manager(Mgr#, m_name, office#)

Department(Dept#, d_name)

Project(Proj#, p_name, startdate, enddate)

Supplier(Supp#, s_name, phone)

Part(p#, quant, date)

Employee(soc_sec,e_name,hiredate,salary, title)

<u>Figure 7.4</u>

attributes) in each entity of Figure 7.4, identifies the entity uniquely; such as soc_sec would an Employee. Notice that the concept of 'uniquely defined' has been applied to both relationships between entities, and within entities themselves.

We are now in a position to define truth functions for the structure between the entities as depicted in Figure 7.3, and within the entities as defined in Figure 7.4.

First we define functional dependencies within entities by setting up attributes as functionally dependent on an attribute (or group of attributes) that uniquely define the entity. The resulting truth functions are as depicted in Figure 7.5.

We will call the attribute(s) on which the other attributes in an entity are functionally dependent the <u>key</u> of the entry.

Notice that Part has been omitted from Figure 7.5. This is because the many-to-many mapping between Part and Supplier, and Part and Project, as depicted by the double-headed arrow in Figure 7.3. To establish functional dependency truth functions for such entities, we include the key attributes of the entities at the <u>tail</u> end(s) of the double-headed arrow, which yields in this case:

$8f(|p#,Proj#,Supp#|,|quant,date|)=1

This same approach would be taken (ie: a double headed arrow) if there were not attribute(s) within an entity that uniquely identified the entity.

```
$8f(|Mgr#|,|m_name,office#|)=1
$8f(|Dept#|,|d_name|)=1
$8f(|Proj#|,|p_name,startdate,enddate|)=1
$8f(|Supp#|,|s_name,phone|)=1
$8f(|soc_sec_|,|e_name,hiredate,salary,title|)=1
```

Figure 7.5

This takes care of intra-entity dependencies, but neglects inter-entity dependencies that are portrayed in Figure 7.3. All we have done thus far is take account of the double-headed arrow of Figure 7.3, but not any other types of arrows.

In order to handle the single-headed arrows we proceed as follows:
Add the key of the entity at the tail to the list of functionally dependent attributes of the entity at the head of the arrow.
For the entities of Figure 7.5, and the interrelations of Figure 7.3, this process generates the entities of Figure

$8f(|Mgr#|,|m_name,office#|)=1

$8f(|Dept#|,|d_name|)=1

$8f(|Proj#|,|p_name,startdate,enddate,Mgr#,Dept#|)=1

$8f(|Supp#|,|s_name,phone|)=1

$8f(|soc_sec|,|e_name , hiredate , salary , title, Dept#, Proj#|)=1

$8f(|p#,Proj#,Supp#|,|quant,date|)=1

Figure 7.6

7.6

Now all that is left to consider is the arrow head with the '='. This type of arrow indicates a one-to-one mapping, or a mutual dependency, and so a mutual dependency truth function is established for the keys of the entities at the tail, and head of the arrow. Thus, from Figure 7.3 we have:

$8m(|Mgr#|,|Dept#|)=1

We now have a set of truth functions that reflect the entity attributes, as well as the interrelations between the entities. Now the SDS, by applying the algorithm specified in Appendix 1, generates the third normalized relations of Figure 7.7.

<u>Implementation of Relations</u>.

The procedure of diagramming the inter-entity relationships as outlined above provides a logical method for establishing the truth functions necessary for the SDS to maintain third normalization.

The next function of the SDS at this stage is to determine the implementation of the relations of Figure 7.7. The decisions to be made are:

- virtualizing and encoding decisions,
- indexing decisions
- factoring decisions

No derived relation, or permanent join decisions can be made at this point since there are no virtual relations, and no new domains have been defined to be included in the data base. Thus all decision variables referring to virtual

RR1(Mgr#,Dept#,m_name,d_name,office#)

RR2(Proj#,Mgr#,p_name,startdate,enddate)

RR3(soc_sec, Proj#, Dept#, e_name, hiredate, salary, title)

RR4(Supp#, s_name, phone)

RR5(p#,Proj#,Supp#, quant, date)

Figure 7.7

(Keys underlined)

relations, and relations with the same key will be 0, resulting in the effect of the appropriate decision rules beeing null.

In order to employ the various decision rules, we need to have values for some of the decision variables. At this point (ie: in definition phase) these values must be user-supplied.

For our purposes, we can deal with some of the aggregations of decision variables, and allow the SDS to split these aggregations into detailed decision variables as needed. All decision variables which do not have user-supplied values, will be 0.

Suppose we know the following about the use of the data base:

a) RR4 is usually accessed on s_name,

b) RR2 is usually accessed on p_name

c) RR1 is usually accessed either on m_name or d_name, equally often on each

d) The enddate of a project (in RR2) is <u>always</u> 3 months after the startdate. (All projects run for 3 months.)

e) title of RR3 has exactly 46 possible values, and is not expected to change. It is also seldom accessed.

For (a),(b) and (c), the SDS should consider indexing the relevent attributes.
For (d), virtualizing of enddate is possible, and for (e) encoding of title is possible.

We will set up the following decision variable values for use in further explanation:

```
$1rd(RR4,s_name)*se* = 10

$1rd(RR2,p_name)*se* = 10

$1rd(RR1,m_name)*se* = 5

$1rd(RR1,m_name)*ce*n = 5

$1rd(RR1,d_name)*se* = 5

$1rd(RR1,d_name)*ce*n = 5
```

All other decision variables have initial values of 0. Other pertinent data is:

```
$5cy(RR1)r = 60

$5cy(RR2)r = 123

$5cy(RR3)r = 2000

$5cy(RR4)r = 75

$5cy(RR5)r = 25000


$6(title)q = 46

$6(s_name)q = 10

$6(p_name)q = 89

$6(m_name)q = 58

$6(d_name)q = 60


$4bfe = 25

$4bfx = 350
```

$4io = 0.0012

$4opc = 0.005

$4t = 1

$4sc = 1.6 x 10**(-5)

7.1 Virtualizing Decisions.

Suppose:

$1rd(RR2,enddate)*s**=1,

$1rd(RR2,enddate)*c**n=1,

$2rd(RR2,enddate)r**=2, and

$2rd(RR2,startdate)u**=5 .

Using decision rule 5.3.1 of Chapter V:

Cost(making domain real), and cost(virtualizing) are both 0, since there is no data yet in the system.

cost(use if real)= (1+1).(123/25) x 0.0012
+ 123 x 0.005
= 0.63

cost(maintaining if real) = 5 x (0.0012 + 0.005)
= 0.03

cost(final) = (123/25 x 0.0012) + 123 x 0.005
= 0.62

cost(use if virtual) = (0.62 x 2)+(0.62 x 2)

= 2.48

Applying the decision rule of 5.3.1, we have:

Make the domain real if:

(0.63 + 0.03) < (2.48).

In this case, the domain would be made real. (Note that the main reason for this is the fact that the domains is used as a qualifier, thus requiring a linear search of the relation to compute, and then test the value of the domain.

7.2 Encoding Decisions.

The candidate here for encoding is 'title' in RR3.

Suppose:

$1rd(RR3,title)***** = 1, and

$2rd(RR3,title)|r,u|** = 2.

Then, applying the decision rule 5.3.2 of Chapter V:

cost(encoding) and cost(decoding) are both 0, since there is as yet no data in the data base.

cost(use if encoded) = 2.(1+2).(0.0012+0.005)

= 0.04

cost(use if unencoded) = (1+2).(0.0012+0.005)

= 0.02

```
cost(extra storage) = ( (4 x 2000)
                             - (6+(8 x 46)+100) )
                           x (1 x 1.6 x 10**(-5))
                     =.12
```

Thus, using the decision rule of 5.3.2, encode the domain if:

(0.04) < (0.12 + 0.02).

In this case, 'title' of RR3 would be <u>encoded</u>.

7.3 Indexing Decisions.

For each of the domains used as qualifiers with a <qualifier type> of 'e', the SDS would evaluate the desirability of creating an index (if one did not already exist) for that domain. If an index exists, the SDS would determine whether it is still needed.

We shall only follow one case here; namely, for p_name in RR2.

Applying the decision rule 5.4 of Chapter V:

```
cost(storage) = (50+(8 x 123)+(4 x 89))
                x ((1.6 x 10**(-5)) x 1)
```

= 0.02

cost(projected use with index) =

(123/89) x 10 x (0.0012+0.005)

+ 10 x (123/350) x 0.0012

= 0.09

cost(projected use without index) =

10 x (((123/25) x 0.0012)+(123 x 0.005))

= 6.21

cost(building index) =

0.005 + 0.0012 x((123/25)+(123/350))

= 0.01

Thus, the decision becomes:

Build an index for the domain if:

(0.02 + 0.09 + 0.01) < (6.21)

which would result in a decision to build an index for p_name of RR2.

No permanent join, or derived relation decisions are made for reasons outlined above.

Once the system has been operational for a while, and values have been generated for the different decision variables, an invocation of the SDS would make similar decisions in a similar way, to the examples above. It would, in addition, make permanent join decisions (where applicable) and derived relation decisions, as specified by decision rules 5.6 and 5.7 of Chapter V respectively.

This scenario has presented a simple application to demonstrate the manner in which the various decision rules would be applied. The reader is invited to experiment with other scenarios and other decision rules in a fashion similar to that employed here.

Conclusion.

What has been presented here is:

- a methodology for pseudo-optimization of a data base for the type of use currently being made thereof. This is done by the SDS.
- a procedure for pseudo-optimaztion of request handling, by the RDS.

The phrase 'pseudo-optimal' is used in preference to the word 'optimal', since the decision rules presented here are largely heuristic, and as such may well not be optimal in the accepted sense of the word.

The SDS is driven by a collection of decision variables (maintained by the RDS) and a collection of truth functions which are used in SDS decision rules. The output of the SDS is the pseudo-optimal, third-normalized data base structure.

The RDS is driven primarily by a set of truth functions, but does make some use of decision variables. The output of the RDS is:

- updated decision variables,
- a collection of set-theoretic operators to best satisfy the request.

The decision rules in both the SDS and the RDS are highly modularized to permit replacement of particular decision rules, and parts of decision rules without effects on other parts of the subsystem. Furthermore, all decision rules are highly implementation specific, and it is envisioned that this will generally be the case, as generalized decision rules may well degenerate into a summation of specific rules, connected by boolean variables. As such, the modularity of the decision rules presented here may be a major feature to allow for easy replacement of those parts of rules that are appropriate for other implementations.

Perhaps the most outstanding feature of the approach taken here is the dynamic nature of both the SDS and the RDS. To date, this has certainly not been true of subsystems used to aid in the design of the data base, and only rarely in the request-handling function. **

In general, queries have had to be stated in a way that inherently specified the procedural steps to be taken in its handling, and structuring decisions have always been made by someone in the position of a data base administrator. This person (or group) might well be aided by some type of decision model, but such structuring, and more importantly restructuring decisions were never made dynamically by the system.

Various algorithms were developed for aiding in the process of optimizing performance, the most major of which is that in Chapter VI for pseudo-optimizing request handling.

This work can certainly not, nor does it, claim to be complete in any sense. It is merely to demonstrate a methodology for approaching the arena of automated decision subsystems. As such, there are many areas into which forays must be made before such subsystems become complete.

** IBM's San Jose Research Center has designed a query language called 'Sequel' which does quite elaborate dynamic request handling optimization. See (8).

Future Research.

Perhaps the first step should be to apply the methodology presented here to other situations. XRM was the only implementation considered here.

There is also a problem that arises from the fact that XRM is physically implemented in a way that is rather analogous to the interface that the user sees. As such, there was little attempt (or need) to separate these aspects of the system. There is, however, a clear need for such a separation and the SDS should be broken down into two distinct parts to handle:

- the logical structure, and
- the physical structure.

It might seem that virtual relations are, in fact, the logical system structure, but further reflection will reveal the fact that real relations can be physically implemented in a variety of ways. XRM treats, and stores each entity as a row, whereas some systems are more column-oriented, in that an entity consists of a value from each column. It is even possible to implement a relational system in a system of the IMS variety.

In this regard, the (third-normalized) real relations used throughout this thesis may, in fact be physically implemented in a number of ways. The SDS should be expanded to reflect this aspect of data management systems.

There were also places in the body of this thesis where the partial inaccuracy of the decision rule was pointed out. These modifications, as well as many other refinements could be made to those rules presented here. It is important, though, to recognize when fine-tuning will yield major improvements, and when the benefits are substantially below the costs of such efforts.

It is felt that the decision rules presented here are of the type that might affect system performance by many orders of magnitude, particularly in cases where usage changes over time. Fine tuning these rules might affect performance by only a few percentage points. Perhaps this should indicate that research conducted in this vein attempt to first accomplish the orders-of-magnitude improvements before any fine tuning is attempted.

1) Donovan, J.J., Systems Programming, Chapter 7, McGraw Hill, 1972.

2) Gries, David, Compiler Construction For Digital Computers, John Wiley & Sons, 1971

3) Codd, E.F., 'A Relational Model of Data for Large Shared Data Banks', Communications of the ACM, Vol. 13, # 6, June 1970.

4) Codd, E.F., 'Further Normalization of the Data Base Relational Model', IBM, San Jose, 1971.

5) Lorie, R.A., 'XRM - An Extended (N-ary) Relational Memory', IBM Cambridge Scientific Center, Cambridge, Ma, Jan. 1974 (IBM Report G320-2096)

6) Codd, E.F., 'Relational Completeness of Data Base Sublanguages', IBM, San Jose, March 1972 (RJ 987)

7) Bachman, C.W., 'Data Structure Diagrams', Data Base (Quarterly News letter of the ACM-SIGBDP) Vol. 1, #2, 1969.

8) Astrahan, M.M., Chamberlin, D.D., 'Implementation of a Structured English Query Language', IBM Research, San Jose, Oct. 10, 1974.

1) Ackermann, R.C., 'An Examination and Modelling of a Prototype Information System', Masters Thesis, Sloan School of Management, MIT, June 1973.

2) Bachman, C.W., 'Data Structure Diagrams', Data Base (Quarterly News letter of the ACM-SIGBDP) Vol. 1,#2, 1969.

3) Bachman, C.W., 'The Data Base Set Concept; Its Usage and Relaization', Honeywell Information Systems, Internal Report, Jan. 31, 1973.

4) Brent, R.P., 'Reducing the Retrieval Time of Scatter Storage Techniques', Communications of the ACM, Vol. 16, #2, Feb. 1973.

5) Buchholz, W., 'File Organization and Addressing', IBM Systems Journal 2, (June 1963) pp 86-111.

6) Burkhard, W.A., 'Some Approaches to Best-Match File Searching', Communications of the ACM, Vol. 16, #4, April 1973.

7) Cardenas, A.F., 'Evaluation and Selection of File Organization - A Model and System', Communications of the ACM, Vol. 16, #9, Sept. 1973. 8) Chamberlin, D.D., Gray, J.N., Traiger, I.L., 'Views, Authorization, and Locking in a Relational Data Base System', IBM Thomas J. Watson Research Center, Dec. 19, 1974 (RJ 1486).

9) Chapin, N., 'A Comparison of File Organization Techniques', Proceedings ACM, 24th National Conference, 1966.

10) CODASYL Systems Committee, <u>Feature Analysis of Generalized Data Base Management Systems</u>, ACM, May 1971.

11) CODASYL Systems Committee, <u>A Survey of Generalized Data Base Management Systems</u>, ACM, May 1969.

12) Codd, E.F., 'A Relational Model for Large Shared Data Banks', Communications of the ACM, Vol. 13, #6, June 1970.

13) Codd, E.F., 'Relational Completeness of Data Base Sublanguages', IBM Research, San Jose, Mar 1972.

14) Codd, E.F., 'Further Normalization of the Data Base Relational Model', IBM Research, San Jose, 1971.

15) Codd, E.F., 'Seven Steps to Rendezvous with the Casual User', IBM Research, San Jose, Jan 17, 1974.

16) Codd, E.F., 'Normalized Data Base Structure; A Brief Tutorial', IBM Research, San Jose, Nov. 1971.

17) Codd, E.F., 'A Data Base Sublanguage founded on the Relational Calculus', Proc. 1971 ACM-SIGFIDET Workshop, 1972.

18) Collmeyer, A.J., Shemer, J.E., 'Analysis of Retrieval Performance for Selected file Organization Techniques', Fall Joint Computer Conference, 1970.

19) Dodd, G.D., 'Elements of Data Management Systems', ACM Computing Surveys 1, June 1969.

20) Donovan, J.J., Systems Programming, McGraw-Hill, 1972.

21) Follinus, J., Madnick, S., Schutzman, H., 'Virtual Information in Data Base Systems', Sloan School Working Paper, Sloan School of Management, MIT.

22) Frank, R.L., Yamaguchi, K., 'A model for a Generalized Data Access Method', National Computer Conference 1974.

23) Ghosh, S.P., 'File Organization: The Consecutive Retrieval Property', Communications of the ACM, Vol. 15, #9, Sept 1972.

24) Hanson, R.J., 'Stably Updating Mean and Standard Deviation of Data', Communications of the ACM, Vol. 18, #1, Jan 1975.

25) Hsiao, D., 'A Formal System for Information Retrieval from Files', Communications of the ACM, Vol. 13, #2, Feb 1970.

26) Huang, J.C., 'A Note on Information Organization and Storage', Communications of the ACM, Vol. 16, #7, July 1973.

27) Langefors, B., 'Some Approaches to the Theory of Information Systems', BIT(3), 1963.

28) Langefors, B., 'Information System Design Computations using Generalized Matrix Algebra', BIT(5), 1965.

29) Lefkovitz, D., File Structures for On-line Systems, Spartant Press, Washington, 1969.

30) Lowe, T.C., 'The Influence of Data Base Characteristics and Usage on Direct Access File Organization', JACM, Vol. 15, #4, Oct. 1968.

31) Lowenthall, E., 'A Functional Approach to the Design of Storage Structures for Generalized Data Management Systems', Ph.D. Thesis, U. Texas, Austin, Aug. 1971.

32) Lum, V.Y., 'Multi-attribute Retrieval with Combined Indices', Communication of the ACM, Vol. 13, #11, Nov. 1970.

33) Lum, V.Y., Yuen, P.S.T., Dodd, N., 'Key-to-Address Transformation Techniques: A Fundamental Performance Study on Large Existing Formatted Files', Communications of the ACM, Vol. 14, #4, Apr 1971.

34) Madnick, S.E., Donovan, J.J., Operating Systems, McGraw-Hill, 1974.

35) McCuskey, W.A., 'Toward the Automatic Design of Data Organization for Large Scale Information Processing Systems', Ph.D. Thesis, Case Western, Jan. 1969.

36) McCuskey, W.A., 'On Automatic Design of Data Organization', Fall Joint Computer Conference, 1970.

37) Mullin, J.K., 'Retrieval-Update Speed Tradeoffs Using Combined Indices', Communications of the ACM, Vol. 14, #12, Dec. 1971.

38) Rothnie, J.B., Lozano, T., 'Attribute Based File Organization in a Paged Memory Environment', Communications of the ACM, Vol. 17, #2, Feb 1974.

39) Sagamang,J.P., 'Automatic Selection of Storage Structure in A Generalized Data Management System', Masters Thesis, UCLA, 1971.

40) Severence, D.G., 'Identifier Search Mechanisms: A Survey and Generalized Model', Computing Surveys, Vol. 6, #3, Sept. 1974.

41) Schachat, I.J., 'A Parameterized Model for Selecting the Optimum File Organization in Multi-attribute Retrieval Systems', Masters Thesis, Sloan School of Management, MIT, June 1974.

41) Shneiderman, B., Scheuermann, P., 'Structured Data

Structures', Communications of the ACM, Vol. 17, #10, Oct. 1974.

42) Shneiderman, B., 'Optimum Data Base Reorganization Points', Communications of the ACM, Vol. 16, #6, June 1973.

43) Siler, K.F., 'A Stochastic Model for the Evaluation of Large Scale Data Retrieval Systems...', Ph.D. Thesis, UCLA, 1971.

44) Stamen, J.P., Wallace, R.M., 'Janus: A Data Management and Analysis System for the Behavioral Sciences', Cambridge Project, Cambridge, Ma.

45) Stocker, P.M., Dearnley, P.A., 'Self Organizing Data Management Systems', The Computer Journal, Vol. 16, #2, 1973.

46) Winkler, A., 'A Methodology for Comparison of File Organization and Processing Procedures for Hierarchical Storage Structures', Ph.D. Thesis, U. Texas, Austin, Aug. 1970.

47) Ziering, C.A., 'Management Information Systems - A Comparison of the Network and Relational Models of Data', Masters Thesis, Sloan School of Management, MIT, June 1975.

This appendix deals with the procedure of third normalization.

The alogorithm presented here is driven by a set of truth functions that detail the functional- and mutual dependencies existing in the data. Notice that computational dependencies are not considered in any of the alogorithms presented here. The issue of computational dependency is not relevent decisions as to which relation a domain belongs in. We proceed now with the algorithm.

1 Apply transitivity to all mutual dependencies
Ie: ∀|a|,|b|,|c| if $8m(|a|,|b|)=1 and $8m(|b|,|c|)=1 then set $8m(|a|,|c|)=1.

Combine all functional dependencies with the same first list, and remove those with duplicate first lists.
Ie: ∀|a|,|c| where $8f(|a|,|b|)=1, $8f(|c|,|d|)=1, and |a|=|c|, set set $8f(|a|,|e|)=1 where |e|=|b|U|d|, and set $8f(|a|,|b|)=0 and $8f(|c|,|d|)=0. 2 Expand functionally dependent domains to include the functionally dependent domains of all mutually dependent (sets of) domains.

Ie: ∀|a|,|b| where $8m(|a|,|b|)=1, $8f(|a|,|r|)=1 and $8f(|b|,|s|)=1:

modify |r| to |r'|, and |s| to |s'| where

|r'|=|s'|= |r| U |s|

This yields: $8f(|a|,|r'|)=1 and $8f(|b|,|s'|)=1.

3 Remove dependencies on partial candidate keys.

(Underlined domains are primary keys; if more than one set of domains is underlined in any one relation, then each underlined set of domains is a candidate key.)

∀|b|,|d| where $8f(|a|,|b|)=$8f(|c|,|d|)=1:

If |b| N |d|=|b| and |a| N |c|=|a| then :

|d|=|d| - |b|, and |x|=|x| - |b| ∀|x| such that $8f(|r|,|x|)=1 and $8m(|r|,|c|)=1.

4 Now set up relations in third normal forms in the following two steps:

4.1 ∀|b|,|d| where $8f(|a|,|b|)=1 and $8f(|c|,|d|)=1:

Does some relation already set up contain both |a| and |b|, or both |c| and |d| ?

yes: then mark that functional dependency as 'done', and continue with a different one. Ie: Find some other

|a|,|b|,|c| and |d|.

(0) <u>No</u>: Is |b| N |d|=Ø ?

(1) <u>Yes</u>: If $8f(|a|,|b|)=1 does <u>not</u> 'have relation', then set up relation RRi(<u>|a|</u>,|b|) and mark functional dependency $8f(|a|,|b|)=1 as 'done' and 'have relation', and continue

(2) <u>No</u>: ∀(|b| N |d|)≠Ø do the following:

Is $8m(|a|,|c|)=1 ?

(3) <u>Yes</u>: set up relation RRi(<u>|a|</u>, <u>|c|</u>, |x|) where |x|=|b|U|d|

Mark the functional dependency $8f(|c|,|d|)=1 as 'done' and both of $8f(|a|,|b|)=1 and $8f(|c|,|d|)=1 as 'have relation'.

(4) <u>No</u>: is |b| N |c|=Ø ?

(5) <u>Yes</u>: set up 3 relations:

RRi(<u>|a|</u>, |b|),

RRj(<u>|c|</u>, |d|), and

RRk(<u>|e|</u>) where |e|=|b| N |d|

If |e| is already in some RRm, m≠i and m≠j, then delete RRk.

Mark those functional dependencies as 'done' and 'have relation'.

(6) <u>No</u>: is |c|=|c| N |b| ?

(7) <u>Yes</u>: there is transitive dependence.

Is $8f(|a|,|b|)=1 'done' ?

(8) No: set |b|=|b'| where |b'|=|b| - |d| and restart step 4.1 for this functional dependency set

(9) Yes: set |b|=|b'| where |b'|=|b| - |d| and: strike all domains |d| from the relation set up for functional dependency $8f(|a|,|b|)=1. Restart step 4.1 for that functional dependency.

(10) No: establish relations:

RRi(|a|, |b|) and

RRj(|c|,|d|)

Mark those functional dependencies as 'done' and 'have relation'.

4.2 ∀|a|,|b| where $8m(|a|,|b|)=1, is there some relation (created in step (4.1) containing both |a| and |b|, eg: (...,|a|,...|b|,...)

(11) No: set up relation RRk(|a|, |b|)

(12) Yes: no action

4.3 For all relations RRi created in 4.1 and 4.2, if $8m(|a|,|b|)=1 for any |a|,|b|∈RRi, then remove |b| from RRi. Examples are presented below to clarify the procedure described above. Decision points in (4.1) and (4.2) above have been numbered for use in the examples that follow. All instances of 'dpn' in the examples mean 'decision point n'. Other notation that will appear in the examples is that for expressing functional and mutual dependencies diagramatically rather than in the form of truth functions.

'->' will imply functional dependency. For example (A,B)->(C) means that C is functionally dependent on A and B.

'<--->' implies mutual dependency. For example, (A,B)<--->(C,D) means that (A,B) and (C,D) are mutually dependent.

example 1

a) (P)<--->(Q,S) or: \$8m(|P|,|Q,S|)=1

b) (P) -> R or: \$8f(|P|,|R|)=1

c) (Q) -> R or: \$8f(|Q|,|R|)=1

Step 1: No action

Step 2: set up (d) \$8f(|Q,S|,|R|)=1

Step 3: Using (b) and (c): |R| N |R|=|R|, and |P|N|Q|=Ø thus, no action.

Using (b) and (d): |R|N|R|=|R| and|P|N|Q,S|=|Q| thus, no action.

Using (c) and (d): |R| N |R|=|R| and |Q|N|Q,S|=|Q| thus: |R|=|R|-|R|=Ø in (1), which means that (d) becomes $8f(|Q,S|,<null>), which must be 0 (See pagexx Chapter IV)

Also: $8m(|P|,|Q,S|)=1, so strike |R| from (b) as well.

We now have:

a) $8m(|P|,|Q,S|)=1

c) $8f(|Q|,|R|)=1

Both (b) and (d) are 0.

Step 4.1: Since (c) is the only functional dependence, |R| N |b|=Ø ∀|b| ∢ (c) take dp1:

RR1(Q, R)

Step 4.1 complete.

Step 4.2: Since $8m(|P|,|Q,S|)=1, and RR1 is the only relation, take dp11 and set up:

RR2(P, Q,S)

Thus have relations:

RR1(Q, R), and

RR2(P, Q,S)

Step 4.3: No action

Example 2

i) (A,B,C)<--->(D,E) or: $8m(|A,B,C|,|D,E|)=1

ii) (D,E)<--->(G,K) or: $8m(|D,E|,|G,K|)=1

iii) (A,B,C) -->(Y,Z) or: $8f(|A,B,C|,|Y,Z|)=1

iv) (G,K) -->(X) or: $8f(|G,K|,|X|)=1

Step 1: Apply transitivity to get:

(v) $8m(|A,B,C|,|G,K|)=1

Step 2: for |a|=|A,B,C| and |b|=|D,E| from (i), (iii) becomes: $8f(|A,B,C|,|Y,Z|)=1 (ie: no change), and get (vi) $8f(|D,E|,|Y,Z|)=1

For |a|=|D,E| and |b|=|G,K| from (ii),

(vi) becomes $8f(|D,E|,|Y,Z,X|)=1, and

(iv) becomes $8f(|G,K|,|X,Y,Z|)=1

For |a|=|A,B,C| and |b|=|G,K| from (v),

(iii) becomes $8f(|A,B,C|,|Y,Z,X|)=1, and (iv) is unaffected.

We now have:

i) $8m(|A,B,C|,|D,E|)=1

ii) $8m(|D,E|,|G,K|)=1

v) $8m(|A,B,C|,|G,K|)=1

iii) $8f(|A,B,C|,|Y,Z,X|)=1

iv) $8f(|G,K|,|X,Y,Z|)=1

vi) $8f(|D,E|,|Y,Z,X|)=1

Step 3: Since |A,B,C| N |G,K|=Ø,

|A,B,C| N |D,E|=Ø

and |G,K| N |D,E|=Ø

no action in step 3.

Step 4.1 Using (iii) and (iv):

get |a|=|A,B,C|, |c|=|G,K|, |b|=|Y,Z,X| and |d|=|X,Y,Z|

|b| N |d|≠Ø, so take dp2. $8m(|a|,|c|)=1 from (v) so dp3.

set up RR1(<u>A,B,C</u>, <u>G,K</u>, X,Y,Z), and mark (iv) as 'done' and 'have relation'. Mark (iii) as 'have relation'.

Using (iii) and (iv): |a|=|A,B,C|, |c|=|D,E|, |b|=|Y,Z,X| and |d|=|Y,Z,X|.

|b| N |d|≠Ø so take dp2. $8m(|a|,|c|)=1 from (i), so take dp3.

Note that (iii) 'have relation', so simply <u>add</u> |c| and |d'|=|d| N |b| to RR1

Ie: get RR1(<u>A,B,C</u>, <u>G,K</u>, <u>D,E</u>, X,Y,Z)

Mark (vi) as 'done' and 'have relation'. Since there are no further functional dependencies that are 'not done', proceed to next step.

Step 4.2: Since for each truth function of the form

$8f(|a|,|b|)=1 (ie: (i), (ii) and (iv)) there exists some relation (viz: RR1) containing |a| and |b|, take decision point (12).

Step 4.3: No action

Thus we have relation: RR1(<u>A,B,C</u>, <u>G,K</u>, <u>D,E</u>, X,Y,Z)

Example 3

i) (A,B,C) -->(D,E,F) or: $8f(|A,B,C|,|D,E,F|)=1

ii) (D,E) --> F or: $8f(|D,E|,|F|)=1

Step 1: No mutual dependencies, so no action.

Step 2: No mutual dependencies, so no action.

Step 3: for |b|=|D,E| and |d|=|F|, |b| N |d|=Ø, so no action.

Step 4.1: From (i) and (ii): |a|=|A,B,C|, |c|=|D,E|, |b|=|D,E,F| and |d|=|F|.

|b| N |d|≠Ø so take dp2.

$8m(|a|,|c|)=0 so take dp4.

|b| N |c|≠Ø so take dp6.

|c| N |b|=|D,E| = |c| so take dp7

(i) is 'not done', so take dp8.

(i) becomes $8f(|A,B,C|,|D,E|)=1, and

(ii) is unchanged.

Restarting Step 4.1:

from (i) and (ii): |a|=|A,B,C|, |c|=|D,E|, |b|=|D,E| and |d|=|F|.

|b| N |d|=Ø so dp1.

Set up realtion RR1(A,B,C, D,E), and mark (i) as 'done' and 'have relation'.

From (ii): |a|=|D,E|, |b|=|F|, |c|=|d|=<null>.

|b| N |d|=Ø so take dp1.

Set up relation RR2(D,E, F) and mark (ii) as 'done' and 'have relation'.

Step 4.2: No action

Step 4.3: No action

So we have relations:

RR1(A,B,C, D,E), and

RR2(D,E, F).

Example 4

i) (A,B) --> C or: $8f(|A,B|,|C|)=1

ii) (D,E) -->(C,F) or: $8f(|D,E|,|F,C|)=1

Note that (A,B)<--/-->(D,E) ie: not mutually dependent.

Step 1: No mutual depandencies, so no action.

Step 2: No mutual dependencies, so no action.

Step 3: No action

Step 4.1: from (i) and (ii): |a|=|A,B|, |b|=|C|, |c|=|D,E| and |d|=|F,C|.

|b| N |d|≠Ø, so dp2.

$8m(|A,B,C|,|D,E|)=0, so dp4

|b| N |c|=Ø so dp5.

Set up relations:

RR1(A,B, C),

RR2(D,E, F,C) and

RR3(C).

Mark (i) and (ii) as 'done' and 'have relation'.

Step 4.1 complete, since all are 'done'.

Step 4.2: No action.

Step 4.3: No action

Thus, we have relations:

RR1(A,B, C),

RR2(D,E, F,C) and

RR3(C).

This concludes the examples.

In the examples and definitions that follow we will use relation names of the form : 'R<i>'. This is for convenience only; any character string may be used for a relation name.

Notation.

R<i> is the name of the i th relation

< means 'is a member of'

|....| implies a <u>list</u>, or <u>set</u> of the items between the '|'s.

c(i) is the cardinality (number of entries) in R<i>

n(i) is the degree (number of domains) in R<i>

d(i,j) is the j th domain of R<i>, j=1,..n(i)

v(m)(i,j) is the m th value of d(i,j), m=1,..c(i)

t(i) is an n(i)-tuple in R<i>

  ie: t(i) (v(a)(i,1),v(a)(i,2),...v(a)(i,n(i)))

    a 1,...c(i)

L(|a|) is the length of list a

Ø is the null set - ie: R<i>=Ø implies c(i)=0

a⊆b means a is a subset of b (a=b is legal)

a⊂b means a is a <u>proper</u> subset of b (a≠b)

∀a means for all values of a

Examples

The following examples will be used throughout this section to explain deifnitions.

```
     (NAME,      SOC_SEC,      PHONE,      DEPT#)
R1=( (Smith,     213-07-1666,  232-1500,    15),
     (Donovan,   621-49-2990,  617-1400,    15),
     (Granger,   413-00-0299,  536-5176,     6),
     (Smith,     839-41-6942,  253-1410,     6))

     (NAME,      SOC_SEC,      PHONE,      DEPT#)
R2=( (Madnick,   217-51-7322,  253-6671,    15),
     (Smith,     213-07-1666,  232-1500,    15),
     (Donovan,   621-49-2990,  617-1400,    15))

     (PERSON,    AGE,      CITY)
R3=( (Madnick,   31        Peabody),
     (Donovan,   34,       Ipswitch),
     (Smith,     23,       Boston))

     (NAME,    PHONE)
R4=( (Smith,   232-1500),
     (Donovan,617-1400))
```

	(PERSON,	AGE,	CITY,	STREET_#)
R5=(	(Madnick,	31,	Peabody,	18),
	(Donovan,	34,	Ipswitch,	43))

Definitions

1) <u>Union</u> Symbol: U

Format: R<i>=R<j> U R<k> (j=k is valid)

c(i)=c(j)+c(k)-c(Rj N Rk)

n(i)=max(n(j),n(k))

R<i>= |t(i) : t(i) ∈ R<j>, <u>OR</u> t(i) ∈ R<k>|

Example: R5 = R1 U R2 would yield:

R5=((Smith, 213-07-1666,232-1500,15),
(Donovan,621-49-2990,617-1400,15),
(Granger,413-00-0199,536-5176, 6),
(Smith ,839-41-6942,253-0410, 6),
(Madnick,217-61-7232,253-6671,15))

2) <u>Intersection</u> Symbol: N

Format: R<i> = R<j> N R<k> (i=j=k is valid)

(Note that if n(j)≠n(k), then R<i>=Ø)

R<i> = |t(i) : t(i)∈j <u>AND</u> t(i)∈k|

n(i) = n(j) = n(k)

Example: R6 = R1 N R2 yields:

R6=((Donovan,621-49-2990,617-1400,15),

(Smith, 213-07-1666,232-1500,15))

3) <u>Difference</u> Symbol: -

Format: R<i> = R<j> - R<k>

(Note: If n(j)≠n(k) then:

n(i)=n(j)

c(i)=c(j)

R<i>=R<j>)

n(i)=n(j)=n(k)

c(i)=c(j) - c(k) - c(R<j> N R<k>)

R<i> = |t(i) : t(i)∈R<j> <u>AND</u> t(i)∉R<k>|

Example: R6=R1 - R2 yields:

R6=((Granger,413-00-0029,536-5176, 6),

(Smith, 839-41-6942,253-0410, 6))

4) <u>Cartesian Product</u> Symbol: X

(Sometimes called a 'Cardinal Product')

Format: R<i> = R<j> X R<k> (j=k is valid)

(Note: if n(j) > 1, or n(k) > 1, then each t(j) (or t(k)) must be treated as a <u>single domain</u>, so that effectively n(j)=n(k)=1.)

n(i)=n(j)+n(k)

c(i)=c(j).c(k)

R<i> = | (v(a) (j,1),v(b) (k,1)) ∀b ∈ k, ∀a ∈ j|

ie: R<i> is a set of ordered pairs with first member from R<j> and second from R<k>.

Example: R5 = R4 X R4 yields:

R5=(((Smith ,232-1500),(Smith ,232-1500)),
((Smith ,232-1500),(Donovan,617-1400)),
((Donovan,617-1400),(Smith ,232-1500)),
((Donovan,617-1400),(Donovan,617-1400)))

5) <u>Projection</u> Symbol: P

Format: R<i> = R<j> P (d(j,l)) , l |1,2,...n(j)|

n(i)=L(l)

c(i)=c(j) (Note that redundent entries are <u>not</u> automatically deleted as proposed in some versions. Use the 'compaction' operator to remove redundent entries.)

R<i> = d(j,l) : l 1,2,...n(j)

Example: R5= R2 P (NAME,PHONE) yields:

R5=((Madnick,253-6671),
(Smith, 232-1500),
(Donovan,617-1400))

6) <u>Join</u> Symbol: *

Format: R<i> = R<j>((d(j,l))) * R<k>((θ,d(k,m)))

θ ::= > | < | = | ¬θ

$l \subseteq | 1,2,...n(j) |$

$m \subseteq | 1,2,...n(k) |$

and d(j,l) and d(k,m) must be of the same data type (ie: must be joinable).

$n(i)=n(j)+n(k)-1$ (<u>no</u> duplication of the join domain when θ is '='. There is duplication when θ not '=', but we ignore that rare case here.)

$c(i)=c(j)+c(k)-c(v(a)(j,l))=v(b)(k,m))$, $a=1,..c(j);\ b=1,..c(k))$

$R\langle i\rangle = |\ d(j,b),d(k,a) \quad \forall b \in j,\ \forall a \in k,$ but $a \neq m$:

$\qquad v(g)(j,l)\ \theta\ v(d)(k,m);\ \forall g \in j,\ \forall d \in k |$

Example 1) R6=R2(NAME) * R3(=,PERSON) yields:

(SOC_SEC, PHONE, DEPT#,NAME, AGE,CITY)

R6=((217-61-7232,253-6671, 15, Madnick, 31,Peabody),

(213-07-1666,232-1500, 15, Smith, 23,Boston),

(621-49-2990,617-1400, 15, Donovan, 34,Ipswitch))

Example 2) R6=R3(CITY) * R4(>,NAME) yields:

(NAME, AGE,CITY, PHONE)

R6=((Madnick, 31, Peabody, 617-1400),

(Donovan, 34, Ipswitch,617-1400))

(Note that this example makes no intuituve sense; it was included simply to illustrate the use of '*' when θ ≠ '=')

7) <u>Composition</u> Symbol: .

Format: R<i> = R<j>(d(j,l)) . R<k>(d(k,m)) l ∈ | 1,...n(j) |

m ∈ | 1,...n(k) |

d(j,l) and d(k,m) must be joinable (ie: of the same data type)

n(i)=n(j) + n(k) -2

c(i)=c(j)+c(k)-c(v(a)(j,l) = v(b)(k,m); a=1,...c(j);
b=1,...c(k))

R<i> = (R<j>(d(j,l)) * R<k>(d(k,m))) P (d(j,b));
∀b∈j, except b≠l

(ie: remove domain d(j,l) on which R<j> and R<k> were joined.)

Example: R5=R2(NAME) . R3(PERSON) yields:

(SOC_SEC, PHONE, DEPT#,AGE,CITY)

R5=((217-61-7232,253-6671,15, 31,Peabody),
(213-07-1666,232-1500,15, 23,Boston),
(621-49-2990,617-1400,15, 34,Ipswitch))

8) <u>Permutation</u> Symbol: M

Format: R<i> = R<j> M (d(j,l)); l = 1,...n(j)

n(i)=n(j)

c(i)=c(j)

The only effect of this operator is to re-order the domains in a relation.

Example: R5 = R3 M (PERSON,CITY,AGE) yields:

R5=((Madnick,Peabody,31),
(Donovan,Ipswitch,34),
(Smith, Boston, 23))

9) <u>Compaction</u> Symbol: C

Format: R<i> = C (R<j>) (i=j is valid)

n(i)=n(j)

R<i> = | t(b) :t(b)≠t(a) ; a≠b | (<u>OR</u>: R<i>=R<j> N R<j>)

This operator simply removes all redundent entries from a relation.

10) <u>Restriction</u> Symbol: R

10.1) Diadic restriction:

Format: R<i>=R<j>(d(j,l)) R R<k>(θ,d(k,m)); l ⊆ |1,...n(j)|
m ⊆ |1,...n(k)|

where: L(l)=L(m), and n(k) <= n(j)

then n(i)=n(j)

θ ::= > | < | = | ¬θ

R<i> = |t(j) : v(a)(j,f) θ v(b)(k,g) ∀f<l, ∀g<m, ∀a<j, ∀b<k
a=1,...c(j); b=1,...c(k) |

Example 1) R6 = R2(NAME,PHONE) R R4((=,NAME),(=,PHONE))

yields

R6=((Smith ,213-07-1666,232-1500,15),
(Donovan,621-49-2990,617-1400,15))

Example 2) R6=R2(PHONE) R R4(>,PHONE) yields:

R6=((Madnick,217-61-7232,253-6671,15),

(Donovan,621-49-2990,617-1400,15))

(Note: t(1) of R6 appears because 253-6671 > 232-1500. the fact that 253-6671 < 617-1400 does not affect this.)

10.2) Monadic restriction:

Format: R<i> = R<j>(d(j,l)) R (θ,d(j,m))

l ⊆ |1,...n(j)|

m ∈ |1,...n(k)|

L(l)=L(m)

θ ::= > | < | = | ¬θ

n(i)=n(j)

R<i> = |t(j) : v(a)(j,f) θ v(b)(j,g), ∀f<l,∀g<m,∀ab < j |

Example: R6 = R10(AGE) R (<,STREET#) yields:

R6=((Donovan,34,Ipswitch,43))

11) <u>Division</u> Symbol: /

Format: R<i> = R<j>(d(j,l)) / R<k>(d(k,m)) ;

l ∈ |1,...n(j)|

m ⊆ |1,...n(k)|

This operator is the inverse of the cartesian product; ie:

(R<j> X R<k>) / R<k> = R<j>

Example: Using R5 of (4) above:

R5 / R4 = R4

Decision Variables.

This appendix lists the decision variables that are used throughout this thesis. They are listed in order of rule# as outlined in Chapter IV.

Rule 1

1) $1rd(i,j)rsen domain j of relation i used as the only equi-qualifier (<qualifier type> 'e') in retrieval request; no joins.

2) $1rd(i,j)rsej same as (1), except join involved in resolving request.

3) $1rd(i,j)rcenn domain j of relation i used as one of several qualifiers, as an equi-qualifier; no joins, and none of other domains used as qualifiers had indexes

4) $1rd(i,j)rceni(trss) same as (3), except some other qualifier had index. 'trss' is size of set resulting from using domains with indexes first.

5) $1rd(i,j)rcejn same as (3) except joins involved in resolving request.

6) $1rd(i,j)rceji(trss) same as (4), except joins involved

in resolving request.

7) $1rd(i,j)rcnnn domain j of relation i used as one of several qualifiers but <u>not</u> as equi-qualifier; no other domains used as qualifiers had indexes, and no joins involved in resolving request.

8) $1rd(i,j)rcnni(trss) same as (4) except domain j <u>not</u> used as equi-qualifier.

9) $1rd(i,j)rcnjn same as (5) except domain j <u>not</u> used as equi-qualifier.

10) $1rd(i,j)rcnji(trss) same as (8) except joins involved

11) $1rd(i,j)rnun domain j of relation i used as unspecified qualifier (eg: ...j='all') no joins involved in satisfying request.

12) $1rd(i,j)rnuj same as (11) except joins involved.

13) $1rr(i)rnun unspecified retrieval from relation i; eg: serial retrieval of each entry in relation. No joins involved.

14) $1rd(i,j)rnuj same as (13) except joins involved in request.

The same set of 14 decision variables is maintained for <requests> 'u' and 'd', and also for <relation type>s 'v' and 'd'.

For <request> 'i', only one decision variable is maintained:

$1rr(i)inun   inserts of entries into relation i; no joins involved.

Rule 2

1) $2rd(i,j)rsn   retrieval of only domain j of relation i; no joins involved in satisfying request.
2) $2rd(i,j)rsj   same as (1) except joins involved in resolving request.
3) $2rd(i,j)rcn   domain j of relation i one of several retrieved; no joins involved.
4) $2rd(i,j)rcj   same as (3) except joins involved in satisfying request.
5) $2rd(i,j)r(<aggr>)sn   retrieval of some single <aggr> of domain j of relation i; no joins involved in request.
6) $2rd(i,j)r(<aggr>)sj   same as (5) except joins involved.
7) $2rd(i,j)r(<aggr>)cn   retrieval of several aggregations, one of them domain j of relation i; no joins

involved.

8) $2rd(i,j)r(<aggr>)cj sames as (7) except joins involved.

9) $2rr(i)ren retrieval of whole entry from relation i; no joins involved.

10) $2rr(i)rej same as (9) except joins involved.

The same set of decision variables is maintained for <request>s 'u' and 'l', and for <relation type>s 'v' and 'd'.

For <request> 'i', only one decision variable is maintained:

$2rr(i)ien inserts of entries into relation i; no joins involved.

Rule 3.

Only one decision variable is maintained of this type:

$3rd(i,j)d(k,m) the number of times relation i joined to relation k via domains j and m respectively.

Rule 4

1) $4io cost per I/O operation

2) $4opc cost per call to XRM

3) $4bfe number of entries per XRM block

4) $4bfx number of index entries per block

5) $4sc cost per byte per day of storage

6) $4t time period since last SDS invocation

7) $4p XRM blocksize

Rule 5

1) $5cy(i)r cardinality of real relation i

2) $5#d(i)r degree of real relation i

3) $5cy(i)v cardinality of virtual relation i

4) $5d(i)v degree of virtual relation i

5) $5cy(k)d(<method>) cardinality of derived relation k. <method> is the method of derivation. If the derivation did not include restrictions, then <method>::=<null>.

6) $5#d(k)d(<method>) degree of derived relation k

Rule 6

$6(j)q number of unique values in domain j

Rule 7

$7r user-supplied response-time weight factor.

Truth Functions.

$8d(i,j) domain j appears in relation i

$8i(j,k) domain k in relation j is inverted. (For virtual relations, $8i(j,k)=0 always.)

$8p(i,j) domain j is one of the primary key domains of relation i.

$8x(i)r relation i is a real relation.

$8x(i)d(<method>) relation i is a derived relation, and <method> is the method of derivation. If <method> did not involve a restriction, then <method>::=<null>.

$8n(i,j) domain j of relation i is mandatory. Ie: a value must be provided for this domain before an entry in relation i will be made.

<u>Note</u> $8n(i,j)=1 ∀j where $8p(i,j)=1. (Primary key domains are mandatory.)

$8u(j) domain j contains unique values (eg: soc_sec_#)

$8r(i,j) same as $8n(i,j) except that it refers to a role name. Also notice that $8r(i,j) is a subset of $8d(i,j) Thus this is a truth function that tests whether a role name is in relation i.

$8_(j)<data type><storage strategy>

<data type>::=<character> | <fixed> | <float> | <vector> | <bit>

<character>::= c

<fixed>::= x

<float>::= f

<vector>::= t(<size>)

<bit>::= b

<storage strategy>::=<virtual> | <real encoded> | <real unencoded>

<virtual>::= v

<real encoded>::= e

<real unencoded>::= u

This set of truth functions is to test the data type of domain j. For exmaple, if $8_(name)ce=1 then domain 'name' is an encoded character string.

$8f(|m|,|n|) is a truth function that tests whether each of the domains in list |n| are functionally dependent on the whole list |m|.

Note 1) List |m| is not a list of all domains on which members of list |n| are functionally dependent. Each n'<|n| may be functionally dependent on some |x|≠|m| also.

2) If |n|=Ø (ie: is empty) then $8f(|m|,|n|)=0.

$8m(|p|,|q|) is a function that tests whether lists |p| and |q| are mutually dependent. Ie: $8f(|p|,PqP)=$8f(|q|,|p|)=1, and also $8m(|p|,|q|) implies $8m(|q|,|p|).

Transitivity also holds: $8m(|p|,|q|)=$8m(|q|,|s|)=1 implies that $8m(|p|,|s|)=1.

$8c(|p|,q)(<function>) is a function which tests whether q (note that q is not a list) is computationally dependent on domains |p|. For example, if domain q is defined as 'q=6.3 * p' then q is computationally dependent on p. (<function>) is the computation required to derive q from the list of domains |p|.

$8od(k) is a truth function set up for a request. It is '1' if domain k appears as one of the object domains in the request.

$8eq(k) is a truth function used in requests. It is '1' if domain k appears as a qualifier with <qualifier type> 'e'.

$8nq(k) is similar to $8eq(k) except that the <qualifier type> is <u>not</u> 'e'.