Matching storage organisation to usage pattern in relational data bases

Izzeldin Mohamed Osman

"وقل ربّ زدني علماً"

طه : الآية ١١٤

"...and say : My Lord! Increase me in knowledge ".

QUR'ĀN XX,114

Matching Storage Organisation to Usage Pattern in Relational Data Bases

by

IZZELDIN MOHAMED OSMAN

B.Sc. (Khartoum), M.Sc. (Bradford)

A thesis submitted for the Degree of

Doctor of Philosophy

of the University of Durham

October 1974

Department of Computing

University of Durham

ABSTRACT

A significant improvement in the performance of data base systems may be achieved by dynamically optimising the storage organisation and the access paths in accordance with the usage patterns.

The principle of defined relations may be employed to ensure that a given relational data base is tuned to match its usage pattern. This thesis describes some original contributions to the solution of the update problem of defined relations.

Some methods of improving the response time without impairing the utilization of disk space have been investigated, and a generalised page replacement algorithm for the management of the data base work space is recommended.

The arguments in this thesis are supported by examples drawn from existing relational data bases. As a whole, the thesis emphasises the benefits of organising the data base in a manner dictated by the activity of its users.

ACKNOWLEDGEMENT

I am grateful to Dr. J. Hawgood for admitting me to the Department of Computing to carry out research for a Ph.D. and for his perpetual encouragement and help.

I wish to express my sincere thanks to my supervisor, Mr. J.S. Roper, for his invaluable encouragement and helpful advice throughout the research period.

The support, encouragement and guidance received from Dr. T.W. Rogers of IBM UK Scientific Centre is gratefully acknowledged, as is the permission of the management of the Centre for the use of their facilities.

I am grateful to Dr. P. Hall of City University for his encouragement and criticism of some parts of this thesis, to Dr. R.G. Casey of IBM San Jose (California) for his co-operation and kindness, and to my colleague, Les Knight, for his co-operation and help in setting up a geological data base.

I have also benefited from the discussions I had with Stephen Todd, Nigel Martin, Ian Clark, Peter Mason and Barry Aldred.

I am thankful for the University of Khartoum for their financial support.

Finally, I am indebted to Mrs. P.A. Croft for typing this thesis.

To my late father

CONTENTS

PREFACE

For the sake of readers who are not acquainted with the terminology of the relational model, Appendix 1 provides a brief explanation of relational concepts.

A brief survey of the evolution of relational data bases follows this preface.

An early version of the experimental prototype relational data base system IS/1.0 was made available to the author by courtesy of the IBM UK Scientific Centre, Peterlee, Co. Durham, for the investigation of data base problems. A brief description of the IS/1.0 system is given in Appendix 2.

Chapter 1 is the analysis of a geological relational data base. The geological references are compiled separately at the end of Chapter 1. In the text Roman script has been used for geological references.

Chapter 2 introduces the concept of defined relations and contains an original contribution to their update problem.

In Chapters 3 and 4 the management of the data base workspace is discussed. The problem is formulated as a generalised page replacement problem.

Chapter 5 discusses the partitioning of data base relations to match the user's queries.

Chapter 6 investigates the possibility of improving the

response time by the judicious choice of secondary indexes.

The arguments in the thesis are supported by examples drawn from existing data bases.

THE STATE OF THE ART

1. The evolution of integrated data base management systems

The rapid technical growth in the field of computers has given greater speeds, the possibility of storing large sizes of information, and increased complexity of processing. The evolution of high level languages and the arrival of direct access storage devices have widened the fields of computer applications and have enlarged the community of users.

This progress in the technology had its impact on computer files. The file is no longer seen as part of the program. The data describing the file is no longer stored in the program but has now been stored in the file itself. Files have become more integrated and are hence the targets of many programs instead of only one program [Senko et al 1971].

With large files processed on-line, new problems and techniques have evolved. The sequential processing of an entire file to access a single record for a transaction is no longer satisfactory. A well-designed file organisation provides a way of immediately processing transactions one by one, thereby allowing the data base to keep an up-to-the-second stored picture of the real world.

In spite of the attention given to the design of information systems, there is little in common between different systems. This is due to the rapid technical growth, the newness of the field and the wide difference in the needs of the people embraced.

2. The description of data bases

In the above-described environment the work on the description of data base systems has taken two general directions [Senko et al 1973] One approach has been to improve descriptions at a gross-feature level. An example of this group is the CODASYL Systems Committee[CODASYL 1971].

Other workers have constructed more detailed system-independent descriptions. More recently with the recognition of the importance of data base systems, useful publications have appeared, for example by [Childs 1968], the Data Base Task Group(DBTG)[CODASYL DBTG 1969],[Codd June 1970],[Engles 1970] and the [GUIDE/SHARE 1970], to mention only a few.

The latter group developed into two schools following two apparently different approaches: the DBTG and the relational approaches.

3. The relational and DBTG (network) approaches

3.1 The DBTG approach

Some of the salient features of this approach are:

(i) Two stages of data definition are required. The first is performed using the Schema Data Definition Language and the second using the Device Media Control Language. The process of data definition may be performed in a series of stages. [Olle September 1973]

(ii) The programmer sees the data only as it is defined in the Data Definition Language by the data base administrator, i.e. the data base administrator

has more control over the mapping to the physical storage.

(iii) It has a network view [Appendix 1,B] together with a network handling technique. Since the relational model does not support networks at the schema level, the DBTG approach is usually referred to as the network approach.

(iv) The programmer navigates his way through the data base using 'one record at a time logic'. [Bachman November 1973],[McGee 1974] and [Olle September 1974]

There are many existing applications patterned after this approach, e.g. by [Phillips 1973] and [Sibley 1974].

3.2 The relational approach

The relational model is based on a sound set theoretic approach to data. The data is conceptually seen in the form of data tables (called n-ary relations or relations).

In order to interact with this data the user needs to know the names of the relations and the domains (attributes or column headings) of interest to him in each relation.

Codd's relational model provides the following advantages [Codd 1974]:

(i) A simple structure consistent with the semantics of the stored information. This makes it possible to use a logically simple language to interact with the information. The relational data sublanguages are high level languages, e.g. SQUARE [Boyce et al

October 1973] and SEQUEL [Boyce et al December 1973].

(ii) A uniform view of data in the sense that there is no distinction between attributes and relationships: both are represented as tuples. Therefore, a small set of operations is required in the data sublanguage.

(iii) It is a complete model, i.e. all data structures commonly employed in data base systems can easily be cast into relational form.

(iv) It possesses data independence [Appendix 1].

Relational data bases can support networks at the subscheme [Boyce et al December 1973].

3.3 The impact of the relational approach on the computer community

When the relational model was introduced, the mathematical aspect of it attracted the attention of those in the computer community who are mathematically oriented. Indeed, it is extremely difficult for a non-mathematician to follow the early literature on the relational model. Some of the terminology was alien to the data processing community which is mostly concerned with data bases. The reaction to this was:

> "The development of the computational aspect of information handling, and computer design itself, has been university-led. However, data bases and data processing have been mainly developed by the practitioners; data processing has grown to immense proportions with the theoreticians lagging far behind. This may be because there is little or no theoretical basis in most data processing systems, because the theoretical basis is trivial, or because the

world has grasped at the computer, which was designed to be a computing engine, and force-drafted it into its role as an information engine, leaving the theoreticians far behind." [Bachman 1973]

When the friends and foes have tested the validity of the claims supporting the rational model, some informal agreement seems to have emerged. This is summarised in the following:

(a) The user's view of the data and the query language offered by the relational model are viable. This agreement is reflected by the suggestions made for using the relational model as a user interface to data bases which are not relational [Olle 1974], [Bracchi et al 1974] and [Dee et al 1973]

(b) Some concern is expressed about the feasibility of efficient implementations of the relational model with all its powerful capabilities. It is not known how much of the system resources can be traded off for the user convenience. This has led to the development of new approaches to some problems which came into existence with the relational model, e.g. the problems of the third normal form, the development of efficient join operators, etc.

However, the proposition of the relational model has stimulated research and investigation in an expanding area of computer science. With the rapidly progressing technology, the increasing number of casual users and the constantly decreasing cost of computer resources, the model that offers the maximum

user convenience is bound to survive.

One approach to improve the performance of data bases is to gear the storage organisation and the access paths to the user requirements. In this thesis the discussion centres round the tuning of storage requirements in a relational data base in a manner dictated by the usage pattern.

Some of the facilities of the relational model are employed to achieve an optimum utilization of the data base resources and to improve the response time.

Automatic methods for the adaptation of the data base structures to the requirements of users in a multi-user environment are suggested. In this respect the thesis is a contribution to the emerging research field of self organising data management systems [Stocker and Dearnley 1973].

The problems treated in the thesis have stemmed from the practical needs of a relational data base system.

Chapter 1

DESCRIPTION OF THE GEOLOGICAL DATA BASE

In order to have a first-hand pragmatic experience and an insight into the operation and problems of relational data bases one needs:

- a large volume of natural real world data
- a group of users who are interested in extracting information from that data for their own benefit.

Such an environment existed in the Department of Geology at Durham University. The vast amount of geological data and the complexity of the queries involved will, to a reasonable extent, fulfil the purpose of a research worker in the field of information systems. In this chapter, I will describe this sample data base in some detail. In later chapters, frequent reference will be made to this data base.

The provision of a data base for geologists is not a new proposition; however, all previous efforts [i,ii] lack data independence.

1. Geological Information Systems

There are vast amounts of collected data in many fields of geology. The present problems stem from the fact that a large proportion of the material is unpublished and with the absence of standards in the nomenclature, duplication in research efforts arises, for example, analyses have been carried out on rock-forming minerals which are already analysed somewhere else. This duplication is attributed to either the lack of communication or to the different names given to the same rock-forming mineral.

Geologists feel that large data banks for geological data will be an emerging basis for international communication in this field[iii].

There are many files of data on rocks (rock files) and data banks in different places tailored to suit the specific requirements of a community of users. Many Departments of Geology in universities have such special purpose systems. An example is the RKNFSYS (Rock Information System)[i] which is a Fortran based system with its own naming and coding conventions. It has its library of programs which process any of the files required by the user.

A similar example is a pilot project on the storage and retrieval by the computer of geological information from cored boreholes in Central Scotland[iv]. This system has its own query language, naming and coding conventions[v].

On the national level the idea of national systems was investigated by geologists in USA and Canada. If this is fulfilled it will spare the potential loss in time, money and achievement resulting from the costly masses of data being allowed to grow in unrelated and uncoordinated ways. Such a national system has been developed in Canada[vi]. The system adopts national conventions. It suffers from the disadvantage that a programmer or a geologist trained in using the system is needed to phrase the queries before they are submitted to the system[vii]. The retrieval program uses Cobol and Fortran5. A similar national system was developed in the USA[viii].

2. The Data Base

The Durham University geological data base was to be set up for a group of minerals known as amphiboles[ix]. Amphiboles represent a small fraction of the recognised rock-forming minerals (Figure 1). They occur commonly in a wide variety of rock types throughout the world. Amphiboles are useful in the interpretation of the mode of formation of their host rocks, and hence their importance for geologists and petrologists.

Many problems arise when considering the analyses and other information on amphiboles. This is due to the absence of a unique attribute of any of the minerals.

"Numerous varietal names have been given to the amphibole minerals, and many names have been introduced to distinguish minerals with minor differences in chemical composition or optical properties. It is not surprising, therefore, that the amphibole group has a surfeit or terminology, much of it adding to the natural difficulties of identifying and naming members of a mineral group in which a wide variety of atomic substitutions are possible." [ix]

Figure 2 shows some sample data. It is ambiguous, complex and lacks a natural unique attribute. As seen from the example, the same name is given to several minerals, or rather several items. Some of the properties have been redetermined by other workers and the results of both determinations are to be available. Many analyses are published jointly by more than one author.

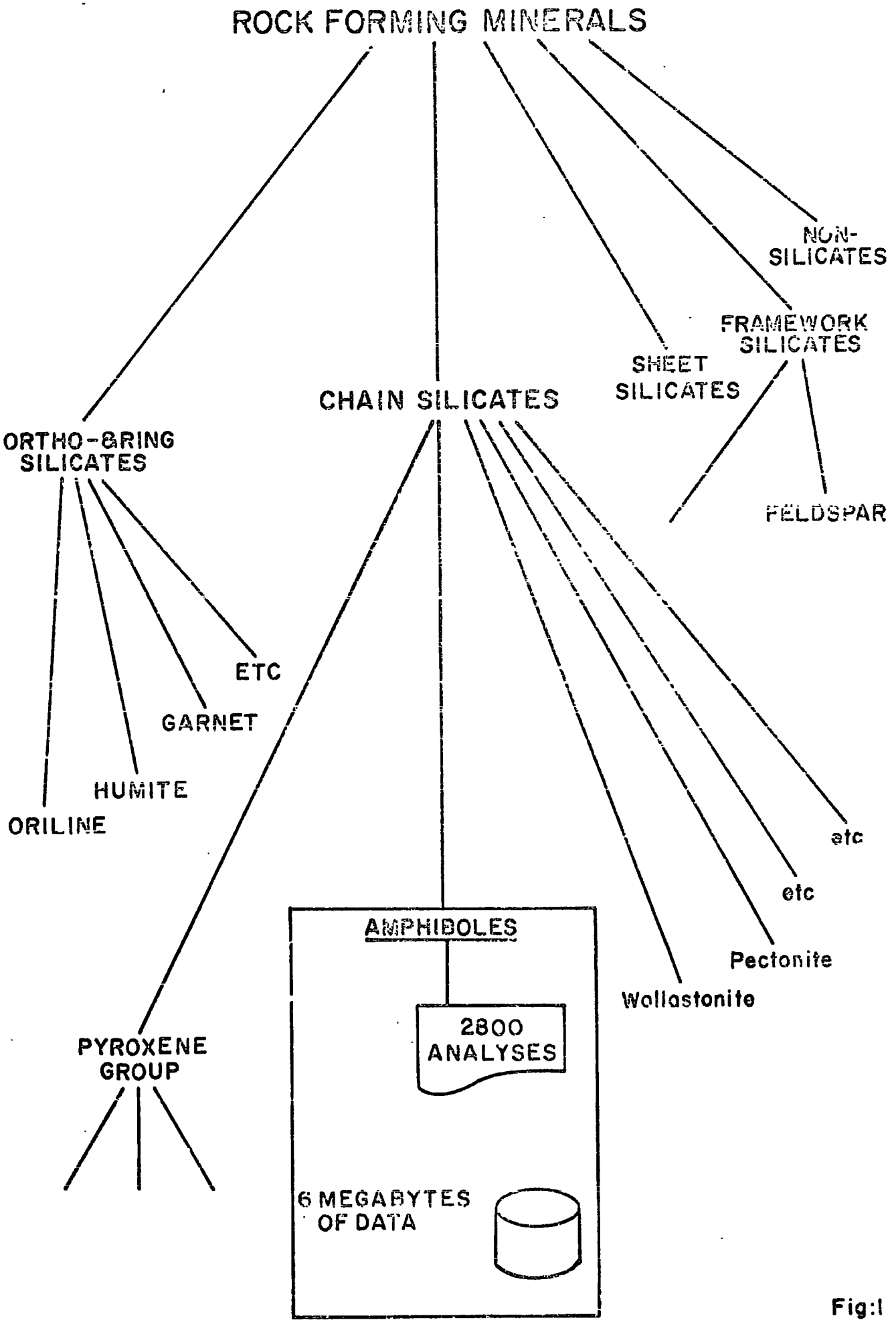

ROCK FORMING MINERALS
NON-
SILICATES
FRAMEWORK
SILICATES
SHEET
SILICATES
CHAIN SILICATES
ORTHO-&RING
SILICATES
FELDSPAR
ETC
GARNET
HUMITE
ORILINE
etc
etc
Pectonite
Wollastonite
AMPHIBOLES
2800
ANALYSES
PYROXENE
GROUP
6 MEGABYTES
OF DATA

Fig:1

1. Mineralname: Hornblende. Found in Cooma District, New Scuth Wales, Australia.

   Analysis published by W. Joplin in 1939. Analysed by W. Joplin.

   Ref: Geol. Soc. Amer. Bull.

   Chemical analysis: SiO_2 50.8%
   TiO_2 0.36%
   Al_2O_3 9.42%
   .
   .
   .
   etc.

   Trace mineral Z_n 6000 parts per million

   Optical properties: α 1.642, β not recorded, γ 1.66, δ 0.017, $2V\alpha$ not recorded, γ :Z=22, D 3.119

   colour: pale green

   pleochroism: α colourless
   β pale yellow-green
   γ pale green

2. Mineralname: Hornblende. Found: not recorded

   Published by Howie & Zussman in 1955. Analysed by Howie & Deer

   Ref: American Mineral

   Chemical analysis: SiO_2 46.8% etc.

   [Chemical analysis redetermined by Deer in 1970. Ref: Geol. Soc. Amer. Bull.]

3. Mineralname: Tschermakite ..... etc.

4. Mineralname: Gedrite ..... etc.

5. Mineralname: Tremolite ..... etc.

6. Mineralname: Tschermakite Hornblende ..... etc.
   (This is a Tschermakite as well as a Hornblende. It is a sensostricto Hornblende and a sensolato Tschermakite[x]).

FIGURE 2

3. Volume of the Information

The information on 2800 amphiboles has been collected. Each of these may contain chemical information, geographical information, bibliographies, etc. It is known that at least twice as much information is available and it is conceivable that this figure could be doubled yet again if more obscure literature sources could be scrutinised.

Associated with each amphibole analysis over 120 attributes (properties) have been recognised. The items of information related to each property are not simultaneously available. The frequency of their availability varies considerably (Figure 3).

4. The Choice of the Primary Key

As seen from the data structure (Figure 2) there is no natural primary key. In many cases no single attribute or a group of attributes uniquely identifies an analysis, e.g. mineralname, author name, reference or date do not necessarily identify a unique analysis. In many cases they may identify a group of analyses for which the author used his own reference code to distinguish between the members of the group. In fact the whole set of data related to a specific analysis is unique.

An artificial key has been introduced to the data. This is a serial number indicating the order in which the analyses were collected for the data base. It uniquely identifies each analysis. The user need not know the value of this number for the analysis he requires. He should, instead, provide the values of some attributes which together uniquely identify

TOTAL NUMBER OF ANALYSES = 2800

2800	analyses	with	Chemical information *
2300	"	"	Geographical information
1000	"	"	Optical information
100	"	"	Colour information
1500	"	"	Reference information
400	"	"	Information on Physical Properties
440	"	"	Pleochroism information
1088	"	"	Trace Elements
1812	"	"	Occurrence information
852	"	"	Description of the techniques used in the analyses

*Each chemical analysis is made up of an average of 10 oxides.

Total number of different oxides recorded to date is 65.

Included in the 2800 entries are 130 duplicate chemical analyses.

FIGURE 3

the piece of information he is seeking. The data base system uses the analysis number (reference number) to tie different properties together.

5. The Formation of Relations

To fit the data into relations the following points were investigated:

(i) The natural relationship within the data items:
The meanings of data items allow us to divide these items into related groups, e.g. the data items related to geographical positions form a group separate from the names of the optical properties, etc.

(ii) The nature of the anticipated queries:
The data base was required to answer queries of the following form:

(a) List all chemical analyses published by Larsen, E.S. (author).

(b) List all Rebeckites (mineral name) with

$Fe_2O_3=0$ and $FeO>0$

or $Fe_2O_3>0$ and $FeO=0$

[N.B: FeO=0 means it has been determined and found to be equal to zero. A different query will be: (List the analyses in which FeO is not determined)]

(c) Is there any analysis from Uganda (geographical position) with CuO>30% (Chemistry) published after 1960 (reference, date).
From which geographical locations were

these samples collected?

(d) List all the Gedrites (mineralname) or those minerals in which Gedrite is used adjectively (e.g. Ferrogedrite) which contain 40-45% of Silica (Chemistry).

(e) What is the name of the mineral whose properties are nearest to the following:

$2V_{\alpha} = 86^{o}$ (Optical property)

absorption formula x>y>z (Pleochroism)

Al_2O_3=20% (Chemistry)

and has a trace of either Zinc or Cobalt.

These queries show clearly that the user sees each analysis mainly as a separate entity and all the various attributes (Chemical, Optical, etc.) as properties linked within that entity, i.e. he sees the data as is listed in Figure 2 and only to a limited extent he looks for properties spanning all the analyses. In relational terms: for the majority of the cases he sees the data in terms of tuples rather than in terms of domains.

In other words there are two different, or rather conflicting, views of the data. One view visualises the data as groups of analyses while the other visualises the data as properties. As more views of the data may exist it seems inefficient to set up relations representing every view of the data. The data base relations would be set up in accordance with one view and hence when new users request

relations expressing other views, these relations will be constructed at a substantial cost. The free storage space available for the data base may be utilised to hold some of the latter relations to avoid the cost of reconstruction. How a relational data base may handle such a situation in an optimum fashion is dealt with in Chapter 2.

(iii) The frequency of availability of data items:
The frequency of availability of attribute values varies considerably. Some attributes belong to a certain group but the frequency of the availability of their values differs very much from that of the values of the other attributes in the group. If the whole group is considered one relation, many of the objects of the relations will have their values marked 'not available'. This leads to a large percentage of 'holes' in the relation. To minimize the number of such 'holes' those attributes whose values are simultaneously available will be grouped together to form a relation.

Having looked at (i),(ii) and (iii) let us consider the following two extremes:

- One relation can be formed with all the attributes as domains (e.g. 120 domains for the data under consideration). The absent values of the attributes are to be marked 'not available'. This will require excessive storage and access time. Figure 4(b) illustrates this type of approach. The information in the example is usually published in this form.

Physics

Ref.#	Property	Value
1	Hardness	16
1	S.G.	3.05
1	Tensile strength	12
2	Hardness	14
2	S.G.	3.2
3	S.G.	3.8

(a)

Physics

Ref.#	Hardness	S.G.	Tensile-strength
1	16	3.05	12
2	14	3.2	NA
3	NA	3.8	NA

(b)

Hardness		S.G.		Tensile-strength	
Ref.#	Value	Ref.#	Value	Ref.#	Value
1	16	1	3.05	1	12
2	14	2	3.2		
		3	3.8		

(c)

FIGURE 4

Possible Formations of Relation Physics

- A large number of small relations with reduced redundancy can be formed as in Figure 4(c). These relations answer some queries very rapidly (e.g. List the minerals whose S.G.>3.2). For some other relations the reduced redundancy form is achieved by having relations in the third normal form. In that case time consuming joins are necessary for some queries.

Somewhere between these two extremes lies an optimum design of the relations. The major factors to be noted at this point are:

(i) how much cpu time can be traded off for disk space, i.e. how far can redundancy be tolerated.

(ii) the possible changes in pattern and frequency of the queries requiring that particular data. It is noteworthy that in the example of Figure 4 it is often not possible to transform one form to another by means of relational operators. Some operators should be defined to perform the transformation. This shows the need for extensible query languages.

It is expected that forms 4(a) and 4(c) will be required by the majority of the queries and because it is easier to obtain (c) from (a), (a) was chosen.

The geological data was converted into relations as follows:

Chemistry	(Ref#,Oxide,Quantity)
Mineralname	(Ref#,Mineralname)
Geography	(Ref#,Country,Province,Locality)

	Relation name	No. of tuples (cardinality)	No. of domains (degree)	Average storage size per tuple in bytes (without overhead)
1	Reference	3350	8	148
2	ANALYSIS	1520	2	16
3	Mineralname	2444	2	52
4	Geography	2208	5	76
5	Occurrence	1812	4	76
6	Optics	1180	15	98
7	Pleochroism	440	5	67
8	Chemistry	29076	3	14
9	Trace	1888	3	9
10	Physics	518	3	14
11	Technique	852	3	35
12	Structure	1024	4	17
13	Colour	77	2	22
14	Redetermination	213	2	4
15	Coexistence	50	2	4
16	Symmetry	95	2	16

FIGURE 5

Specific Description of the Relations

Relation name: Chemistry

no. of tuples = 32000

no. of domains = 3

Domain Name	Type	"Average" Size	Value Distribution
Ref#	Integer	2 bytes	2500 values ranging between 1-2500 each value, occurring at an average of 12 times
OXIDE	Character	8 bytes	65 values ranging from A-Z. The frequency of occurrence of values varies considerably, i.e. between 0 to 2400
QUANTITY	FLOAT	4 bytes	about 30000 values ranging between 0.0-100.0. Values generally occur once

Queries:

The queries which were answered using the above relation were as follows:

. Given Oxide and Ref#, retrieve quantity 23%

. Given Ref#, retrieve analysis 59%
(To be joined on equality of Ref# with another relation)

. Given Oxide and a Quantity limit, retrieve the tuples 18%

FIGURE 6

Description of the Contents of a sample Relation

Optics	(Ref#,Alpha,Beta,Gamma,....,....)
Pleochroism	(Ref#,Formula,X,Y,Z)
Colour	(Ref#,Colour)
Physics	(Ref#,Property,Value)
Trace	(Ref#,Element,Value)
Reference	(Ref#,Author name,Analyst name,Date,....)

The primary key is underlined

[For details of the size of relations, see Figure 5]

In the above relations none of the domains has elements(objects) which are themselves tuples(sets). This ensures that all the relations are of the first normal form [Codd August 1971, Codd November 1971]. Updates (deletions, insertions and change of value) are convenient because the domains of the relations are chosen such that any natural piece of information fits in one relation and so in most of the cases each deletion and change in value affects only one tuple.

Repeated groups were not eliminated in all the cases, e.g. in relation Reference in Figure 7(i) when a reference has more than one author we have a tuple for each author, i.e. the journal name is repeated with each author's name, which leads to the obvious redundancy. The other alternative is to have two normalised relations as in Figure 7(ii) and 7(iii). However, normalisation has the following disadvantages:

(i) an extra relation is needed to get rid of each repeating group. If in Figure 7(i) the queries were to require a search on ANALYSER (e.g. retrieve the tuples where ANALYSER=DEER), then the original relation would have been normalised, giving the three relations (ii), (iv) and (v). This means

that the data base system has to cope with a larger number of relations even if they are not required individually by the users. Accessing a larger number of relations makes the queries more complex.

(ii) the considerable number of join operations when the original relation is needed, i.e. eliminating repetitions avoids redundancy at the expense of cpu time due to joins. This is particularly inefficient when there are no statistics available showing the probability of a join being required.

(iii) the data departs from the natural form in which it originally occurs and the user finds it difficult to visualise multivalued attributes, unless normalisation is only used for storage and update purposes and is transparent to the user.

There was no need to eliminate transitive dependencies because they do not exist among the attributes of the data.

6. Data Capture, Preparation and Validation

When the data was collected the mineralnames were preserved and no coding was performed. This is more convenient for the user. However, the relation Mineralname maps a mineralname into the corresponding reference number, i.e. coding of mineralname is internal to the data base and is transparent to the user.

The remaining part of this section is discussed in further detail in [x].

REFERENCE

Ref#	AUTHOR	ANALYSER	DATE	JOURNAL etc.
1	HOWIE	DEER	1955	American Mineral
1	ZUSSMAN	DEER	1955	American Mineral
2	JOPLIN	JOPLIN	1939	Geol. Soc. Amer. Bull.
3	EVANS	HOWIE & DEER	1968	Amer. J. NSCI
4	LASNIER	'NA'	1969	Contr. Mineral & Petrol
4	FORESTIER	'NA'	1969	Contr. Mineral & Petrol
5	CARMICHAEL	CHAPERLIN	1970	J. Petrology

(i)

AUTHOR

Ref#	AUTHOR
1	HOWIE
1	ZUSSMAN
2	JOPLIN
3	EVANS
4	LASNIER
4	FORESTIER
5	CARMICHAEL

(ii)

REFERENCE

Ref#	ANALYSER	DATE	JOURNAL etc.
1	DEER	1955	American Mineral
2	JOPLIN	1939	Geol.Soc.Amer.Bull.
3	HOWIE & DEER	1968	Amer. J. NSCI
4	'NA'	1969	Contr. Min. & Pet.
5	CHAPERLIN	1970	J. Petrology

(iii)

ANALYSER

Ref#	NAME
1	DEER
2	JOPLIN
3	HOWIE
3	DEER
5	CHAPERLIN

(iv)

REFERENCE

Ref#	DATE	JOURNAL etc.
1	1955	American Mineral
2	1939	Geol. Soc. Amer. Bull.
3	1968	Amer. J. NSCI
4	1969	Contr. Mineral & Petrol
5	1970	J. Petrology

(v)

FIGURE 7

Number of Relations	16
Size of Data Base	6×10^6 bytes
Type of Access	Single User (Batch) pseudo-terminal (Batch) 2741 terminal
Number of Queries	25 Queries/week (while in existence) Total number = 500
System Configuration	360/44 280K Core 3-2314 Disk 2 Magnetic Tapes (Backups)

FIGURE 8

The Geological Data Base System

7. The IS/1.0 Geological Data Base System

The geological data base has been implemented using the IS/1.0 system (Figure 8). The IS/1.0 system is a general purpose information system based on relational algebra. A brief description of the IS/1.0 system and language is in Appendix 2.

In this section I will discuss the actual queries that were submitted to the geological data base because they vary from the anticipated queries. The discussion is independent of the query language and is based on the operations that the relational system carries out. These operations may not be transparent to the user. Before discussing the queries, however, I will site an example of a user extension for the IS/1.0 language to handle a particular type of query.

7.1 User extensions

In IS/1.0 user extensions are normal PL/1 routines which are interfaced with the system via system macros. As an example, consider a user-written TABULATE function.

TABULATE operates on n relations of degrees $d_1, d_2, \ldots, d_n$ and a given domain common to all n relations to produce a new relation of degree $d_1+d_2+\ldots+d_n-(n-1)$, i.e. it forms a relation having the domains of the input relations without repeating the common domain. All the occurrences of the common domain are included. When one of the input relations does not contain a certain occurrence of the common domain, null objects, represented by blanks, are inserted in the tuple.

In Figure 9 an example of the operation of the TABULATE routine is shown. The routine takes as input the names of the three

COLOUR

REF#	COLOUR
1	BROWN
3	GREEN
2	STRAW

OPTICS

REF#	α	β	γ
1	20	100	60
3	80	70	29
4	50	59	72

CHEMISTRY

REF#	OXIDE	QUANTITY
3	SIO2	40
2	FEO	5
4	SIO2	42
4	AL2O3	8
4	FEO	2

```
TABULATE(COLOUR,OPTICS,CHEMISTRY) ON (REF#);
  LIST;
```

yields:

REF#	COLOUR	α	β	γ	OXIDE	QUANTITY
1	BROWN	20	100	60		
3	GREEN	80	70	29	SIO2	40
2	STRAW				FEO	2
4		50	59	72	SIO2	42
4		50	59	72	AL2O3	8
4		50	59	72	FEO	2

FIGURE 9

Example of a User Extension

relations to be tabulated, i.e. COLOUR, OPTICS and CHEMISTRY, and the common domain REF#. The result of the tabulation may be stored back in the data base under a new name.

Though space and time consuming, TABULATE has been a useful function for the geological application.

7.2 Queries

The queries submitted to the data base are grouped into five major types. Examples of these types and their frequency of distribution are shown in Figure 10. These types are influenced by the choice of the underlying relations, e.g. some queries of types 2 and 3 would have belonged to type 4 if the relation Reference [Figure 7] had been in the third normal form, because two separate selections followed by a join would have been necessary.

The queries may be classified according to the complexity of the structure of the query as follows:

Simple queries: These are the queries which interrogate only one relation at a time. These are made up of types 1,2 and 3 in Figure 10. They constitute 37% of all the queries on the geological data base. Indeed a different design of relations would have led to the variation of the frequency of such queries. As redundancy increases, queries tend to become simpler and vice versa.

Moderate queries: These queries interrogate two relations or more at the same time. They constitute queries of type 4 in Figure 10, which account for 58% of all the queries. If, however, more redundancy is allowed in the data base, then

Type	Queries	Frequency
1	From a given relation retrieve those tuples for which the value of one domain is given. e.g. Has the element niobium (Nb) ever been recorded in an amphibole i.e. From relation TRACE retrieve tuples for which ELEMENT[domain(2)]=NB	20%
2	From a given relation retrieve those tuples which have a given value of one domain and a given value of another domain. e.g. List the analyses published by AKOI in 1970 i.e. From relation REFERENCE retrieve tuples for which AUTHOR=AKOI & DATE=1970	10%
3	As (2) but disjunction replaces conjunction	7%
4	Retrieve certain tuples of a relation and join these to certain tuples of another relation. e.g. Obtain the optical properties and the chemical properties for each analysis of the mineral TREMOLITE From relation MINERALNAME retrieve the tuples for which NAME=TREMOLITE Form a projection on REF#[domain 1] and assign the resulting relation to relation R Join R with CHEMISTRY on the equality of REF# and assign the resulting relation to S Join S with OPTICS on the equality of REF#	*58%
5	From a given relation retrieve the tuples for which a domain has one of successive values. Given one value and an increment, i.e. a selection process for which the value to be selected changes dynamically e.g. a. YEAR=1960 b. Select analyses published in DATE=YEAR c. Perform operations as in queries types 1,2,3 or 4 d. If Year=1974 then STOP e. Year=Year+1. goto b.	4%
UPDATES	No. of updates per 500 queries = 23, i.e. about 5%: 3% insertions, 1% deletions and 1% changes in value	

*16% queries with a single join, 28% with two joins, and 14% with more than two joins

FIGURE 10

most of the resulting join operations will be avoided. This would have led to the diminishing of queries of this class and the increase in the number of simple queries.

Complex queries: This class of queries involve loops and control variables. The programmer finds it difficult to track the stages of such queries if the data base query language lacks built-in counts and loop facilities. Queries belonging to this class are time consuming although they account for only 4% of all queries.

The queries of types 4 and 5 take less cpu time if the objects of the requested domains are ordered according to their value. This demonstrates the need for the consideration of sorting in relations. The sorting should not necessarily be added to the set primitives like union, intersection etc., but the concept of ordering the elements of a set according to their value must be accepted as a practical necessity for efficient applications.

Sorting is important in the following cases:

(1) In removing duplicates (Purging)

Consider the query: In relation Chemistry, how many different oxides were analysed? i.e. to find the number of different object values in a domain.

If the operators of the relational data base perform according to their standard mathematical definition, then the result given by any relational operator is a set (i.e. without duplicates). Thus, this query may be answered as follows:

Form a projection of relation Chemistry on domain Oxide. Assign the result to a one domain relation R.

The cardinality of R is the answer to the query. However, if the relational operators are not standard, e.g. IS/1[Notely 1972] where the duplicates are not removed by the projection, then the answer is obtained by first performing the difference of null and R and then finding the cardinality of the resulting relation. To remove the duplicates in the projection or to obtain the difference requires $\frac{m}{2}(n+1)-n$ object comparisons where m is the cardinality of R and n is the number of the different values in the domain [see Appendix 3].

If relation R is sorted, only m comparisons are required. It pays to sort R whenever such a query comes up if $(n > 2.77\log_2 m)$ [for proof see Appendix 3]. This is true in the case of the above example (n = 65, m is just under 32000).

To answer the query: How many authors have published works on amphiboles, it also pays off to sort domain Author of relation Reference (for domain Author n=1317 and m=3350).

(2) In Joins:

Almost all the joins involved in queries of type 4 are joins on the equality of the primary key (Ref#). If all the relations are stored sorted on the primary key, this will reduce the number of object comparisons from $(m_1 \times m_2)$ to $(m_1 + m_2)$ or less; where m_1 and m_2 are the cardinalities

of the relations to be joined.

The overhead due to sorting is tolerable because sorting is done only once for each relation. The overhead due to merging is tolerable if the proportion of the updates is small compared to the number of queries (retrievals) as in the case of the geological data base.

Reducing the join time is essential because join is an operation characterising the relational model and because having read the literature on the relational model the user will find it tempting to store his relations in the third normal form and have them joined whenever necessary. The reduction of the join time has been the concern of relational data base implementors because it has proved to be a performance problem in the implementation of at least one current relational data base[Todd 1974].

(3) Sorting is also essential for the efficient performance of some user-written functions and application programs such as the previously described function: TABULATE.

8. Final Remark

Some of the characteristics of the geological data base may be similar to those data bases established for research and scientific information.

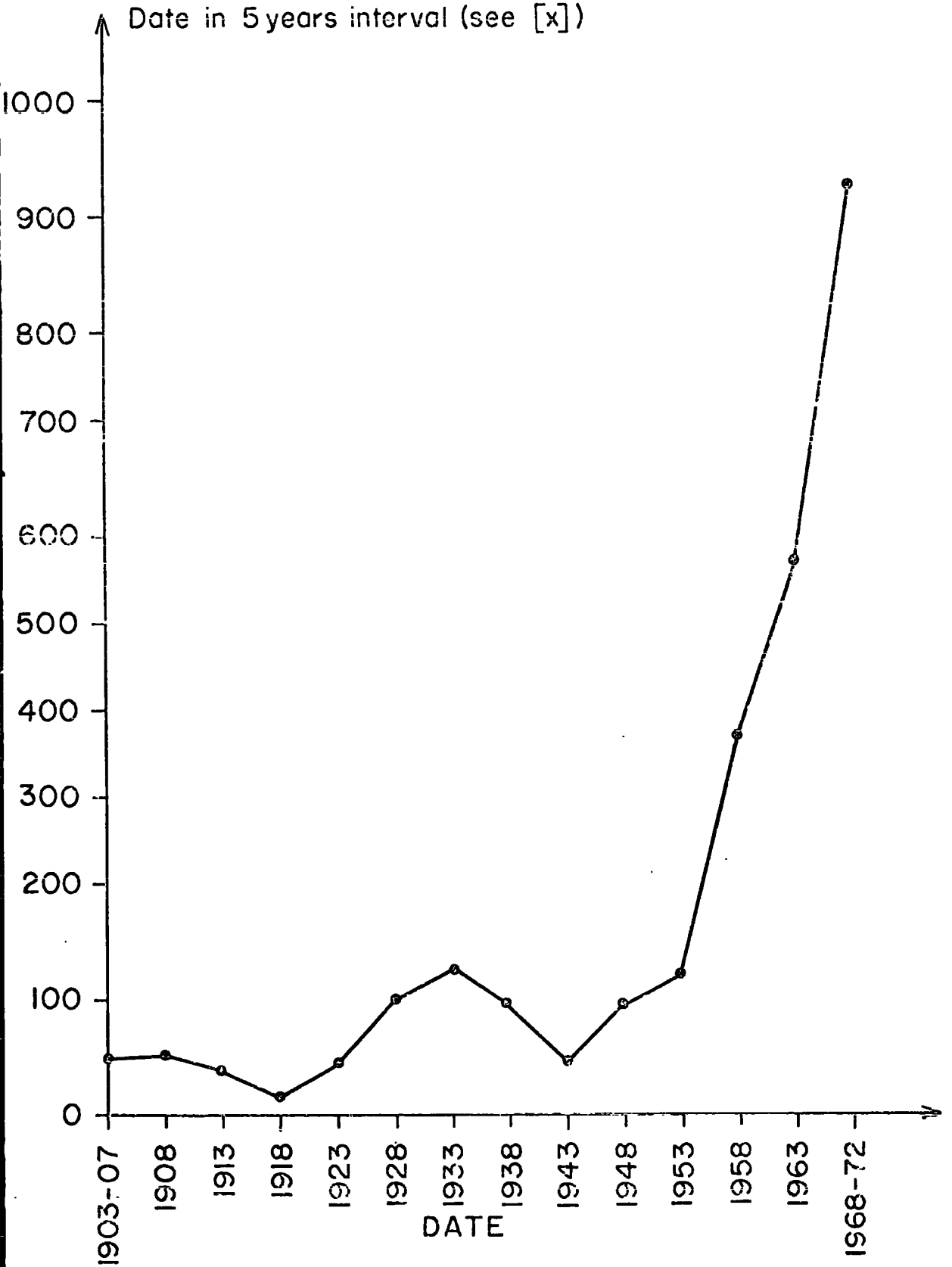
One interesting result, however, is shown in Figure 11. The

number of published analyses is exponentially increasing with time apart from two kinks marking the first and the second world wars. This demonstrates the following:

(i) the necessity for harnessing the computer in processing the ever-increasing volumes of information.

(ii) the importance of the judicious choice of designs for files so that they are robust enough to cope with the huge number of additions.

Fig II Number of Amphibole published analyses



Geological References

[i] CHAYES, Felix — "Rock Information System", Geophysical Laboratory, 2801 Upton Street, N.W., Washington D.C. 2008

[ii] CUTBILL, J.L. WILLIAMS, D.B. — Package for Experimental Data Banking, Systematics Association Special Volume No.3, "Data Processing in Biology and Geology", edited by J.T. Cutbill, 1970, pp.105-113

[iii] BURK, C.F., Jr. — "Computer Based Geological Data Systems: an emerging Basis for International Communication", Proceedings of the 8th World Petroleum Congress (1971)

[iv] HMSO — Report No.71/73, Natural Environment Research Council, Institute of Geological Sciences. Also Report No.71/15

[v] GILL, Elizabeth M. — "Geological Data Banks", Atlas Computer Report 1972

[vi] BURK, C.F., Jr. — "The National System for Storage and Retrieval of Geological Data in Canada", Geophysics Information Society Proceedings, vol.1, 1969, pp.1-7

[vii] DRUMMOND, A.D. — "Development of a Geological Data Retrieval System", Western Miner, February 1969, vol.42, No.2, pp.47-50

[viii] CLARKE — "Automatic Data Processing of Geological Literature by the United States Geological Survey", Geoscience Information Proceedings, vol.1, 1969, pp.8-12

[ix] DEER, HOWIE, ZUSSMAN — "Rock Forming Minerals", vol.2, p.203, Longmans 1972

[x] KNIGHT, J.L. — "The Classification and Interpretation of Amphiboles", Unpub. Ph.D. thesis, Durham University (UK), 1974

Chapter 2

ON DEFINED RELATIONS

1 INTRODUCTION

Some relational data base systems such as IS/1 [Notely 1972] and SEQUEL [Boyce & Chamberlin 1973] allow the user to create his own relations as subsets of the main data. With such a facility the user may impose his own view of the data in the collection. If a large number of relations are created, disk storage space problems and other maintenance difficulties can arise.

The concept of defined (derived or implied) relations is introduced as a storage management measure. Defined relations are subsets of the data that are not assembled physically until they are requested by a query. The user submits to the system a definition of these relations expressed in terms of data base relations using relational operators.

e.g. A user who is interested in the minerals containing Al_2O_3 may issue the following IS/1.0 instruction:

```
DEFINE(MINERALS_WITH_AL2O3);
LOAD(CHEMISTRY);
SELECT(OXIDE=AL2O3);
END;
```

[In the following discussion this definition is expressed as Define Minerals_with_Al2O3=Chemistry:Oxide=Al2O3;]

The system decodes the definition and stores it in a retrievable form. The defined relation will then be available to the user at the same logical level as the other data base relations. As far as the system is concerned, the defined relation remains a stored definition (i.e. implicit) until it is requested by a query. The implicit form takes negligible disk space.

When the defined relation in its implicit form is referenced by a query, e.g.

```
LOAD(MINERALS_WITH_AL2O3);
SELECT(QUANTITY>8);
LIST;
```

it is then created, i.e. made explicit, by carrying out on the stored data the operations indicated in the definition, e.g. to create the relation MINERALS_WITH_AL2O3 the tuples of relation Chemistry are accessed and those matching the selection criterion are written back to disk as tuples of the relation MINERALS_WITH_AL2O3.

The explicit form stays in the data base and the relation will not be recreated if it is requested by another query.

The processing cost to create the relation may be very high. It is a function of the use made by the various parts of the computer system. This cost mainly represents the cpu time and the i/o time spent in processing the definition and assembling the relation. The cost varies with the complexity of the definition and the size of the relations involved in

the definition. The explicit form may take a substantial amount of disk space. However, if the implied relation in its explicit form is requested by a query, no additional cost is incurred.

At some stage in the process of creating and querying relations the available data base space may be consumed. One or more explicit relations will then have to be made implicit in order to free space for other requested relations. A replacement algorithm is needed to decide which relation is to be made implicit. This is discussed in detail in chapter 3 and is followed by the justification of defined relations on a cost basis in Chapter 4.

The definition of the defined relation may contain user-written functions or application programs.

2 THE POTENTIAL ADVANTAGES OF DEFINED RELATIONS

(1) Virtual storage:

The user is freed from worrying about the constraints of disk storage space. He defines as many relations as he may want and as far as he is concerned there is an infinite 'virtual' disk storage space.

(2) Improvement of the response time:

These defined relations are very useful in providing the answers for recurring queries (e.g. the above implied relation MINERALS_WITH_AL2O3 answers the query: which minerals contain Al2O3).

Some users may have narrow needs such that their requirements can be more efficiently satisfied by a set of pre-structured queries in the form of implied relations.

In both of these aspects, reprogramming and compilation times for queries are saved. In addition, for some queries the time to recreate the relation answering the query is also saved. The response time will thus be improved.

(3) Self-optimization of data structures:

Consider the following 4-relation data base:

```
Relation Citizens(Name,Address,Status,Income,No.of
                                        Dependants);
Define Tenants=Citizens:Citizens(status)=Tenant;
Define Landlords=Citizens:Citizens(status)=Landlords;
Define Others=Citizens:Citizens(status)=Others;
```

The relation Citizens is permanently stored (base relation) while the three other defined relations, Tenants, Landlords and Others, will be physically materialised when requested.

Alternatively, the relations Tenants, Landlords and Others may be stored permanently (i.e. made base relations) and the relation Citizens defined as follows:

```
Define Citizens=Tenants ∪Landlords ∪Others
```

By monitoring the usage of these four relations the data base system is able to determine the best strategy under given conditions.

This facility gives the data base system a choice between the alternative methods of storing the same data which enables the system to self-optimize its data structures. This is discussed in detail in chapter 5.

(4) Indexes for domains of relations may be defined as implied relations. This improves the response time and may lead to the optimization of data structures. This is also discussed in detail in chapter 6.

(5) In batch data base systems the system can look at and reorder the queue of requests to take care of the following:

   (i) cut down the processing time by making a relation explicit and grouping together all requests that use this relation.

   (ii) define some relations to cater for repeated queries.

The main disadvantage of defined relations is that they delude the user by giving him different estimates of the computer time required to answer his query. As relations switch between being explicit and implicit the processing time of the query will be greatly influenced by the state of the defined relations.

However, the seriousness of this disadvantage may be offset by

providing the user with a maximum estimate of the computer time required to answer his query.

3 THE MANAGEMENT OF DEFINED RELATIONS

In a data base with a defined relations capability, some of the system operators have to be adapted to take care of the hierarchical structure and the dependencies that exist between the relations. Some possible application of defined relations lead to logical problems which cannot be ignored [Notley 1972]. In the following discussion, recommendations and algorithms are suggested to take care of the following:

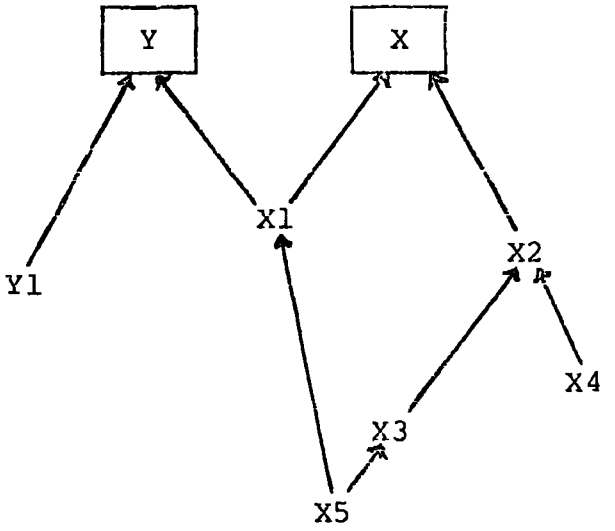
(1) the update problem
(2) the deletion of relations from the system
(3) the definition of relations:
    (i) circular definitions
    (ii) redefinition of relations
    (iii) assignment of defined relations to relations

All these recommendations and algorithms are the independent contribution of the author.

In the examples the normal set notation ($\cap$ for intersection, $\cup$ for union etc.) is used for relations.

3.0 OBJECTIVES

In the following algorithms we are not concerned with search algorithms which get at tuples and carry out the actual operations. We are, however, concerned with the algorithms that deal with the logical part of the problem and hence pave the way for other algorithms to care for the particular details in particular implementations.



Y and X are base relations
Y1 is defined on Y
X1 is defined on X and Y

Figure 2.1

In figure 2.1 a hierarchy of defined relations is shown. In such a hierarchy the user may not be aware of the relationships and the dependencies that exist among the data base relations. It is therefore advantageous to provide the facilities by means of which the user can perform all the operations on a defined relation without restrictions. Ideally, the user should treat the defined relations in exactly the same way as any other base relations. The restrictions force the user to be aware of the dependencies between relations. That is, the system should possess a higher degree of data independence. It is therefore desirable to have the user activities (queries, application programs, updates etc.) independent from the logical representation, the access path and the level in the hierarchy of the data. The user need not know whether a relation is a base or a defined relation, is implicit or explicit, nor will

he need to know whether it is an existing relation or a mere collection of pointers to some other relations.

However, giving all this freedom to the user is nice as long as the consistency of the definitions and the defined relations is preserved. Hence, the objectives of the following algorithms are:

(i) to give the user the maximum freedom (a high degree of data independence).

(ii) to preserve the consistency of the information in the data base.

3.1 UPDATES

Updates can be divided into:

(a) insertions

(b) deletions

(c) changes in object values

The changes in object values are conceptually taken as deletions followed by insertions.

An insertion is seen as a union of the tuples to be inserted (i.e. the updating relation), and the relation to be updated.

A deletion is logically seen as the difference of the relation to be updated and the updating relation.

The Update algorithms

Let us divide the updates into two:

(1) update at higher levels

(2) update at lower levels.

Referring to figure 2.1, updating X or Y is an update at a higher level with respect to X1, X2 and Y1, X1 respectively. Updating Y1, X5 or X4 is an update at a lower level. Updating X1 is an update at a higher level with respect to X5 and an update at a lower level with respect to X and Y.

3.1.1 UPDATE AT A HIGHER LEVEL

In this type the update is filtered down and reflected on all the relations defined on the updated relation. The corresponding definition is applied to the updating relation successively at each level.

Insertions

Examples 1 and 3 illustrate this type of insertion. The insertion is essentially carried out at each level as follows:

(a) The definition is applied to the updating relation (the relation containing the tuples to be inserted). The resulting relation is the updating information which will be passed to the following levels.

(b) If the relation to be updated is explicit, the updated form will be the union of the relation and the updating relation resulting from (a).

The exception to the above rule is the definition containing the difference operator when the second relation is to be updated. Example 2 illustrates this type of definition. In this type the insertion is carried out as follows:

Example 1 (Insertion)

Relation Y

40	A	C
8	T	D
3	L	S

Relation X

20	B	G
8	T	D
14	N	M
3	L	S
4	A	C

Relation X1

8	T	D
3	L	S

Relation X3

8	T	D
2	Q	R

Relation X5

3	L	S

Definitions: X1 = Y intersection X

X5 = X1 difference X3

Relation I (the updating tuples to be inserted in relation Y)

4	A	C
5	M	D

After the update:

(i) Y = Y∪I

Y

4	A	C
5	M	D
40	A	C
8	T	D
3	L	S

(ii) I = I intersection X

X1 = X1∪I

I

4	A	C

X1

4	A	C
8	T	D
3	L	S

(iii) I = I difference X3

X5 = X5∪I

I

4	A	C

X5

4	A	C
3	L	S

Example 2 (Insertion)

Suppose in Example 1:

definition:

(iii) X5 = X3 difference X1

(iv) X6 = project [1] (X5)

After X1 is updated:

X1			X3			X5		
4	A	C	8	T	D	2	Q	R
8	T	D	2	Q	R			
3	L	S	3	S	T			

Suppose I is

4	A	C

X6

2
3

(iii) (I = I) the definition contains difference and the second relation is to be updated.

X5 = X5 difference I

I

4	A	C
2	Q	R

X5

3	S	T

[Alternatively,

(iii) I = X3 ∩ I

X5 = X5 difference I]

Now I is the relation with the tuples to be deleted. The update continues as a deletion operation.

(iv) I = projection [1] (I)

I

4
2

X6 = X6 difference I

X6

3

Example 3 (Insertion)

Relation X5		
8	T	D
3	S	Q

Relation X3		
8	T	D
2	Q	R

Relation Y as in Example 1

Relation X1 is implicit

Relation I (updating tuples)

4	A	C
5	M	D

X1 = Y: select Y[1]<20

X5 = Project([1],[3],[5])(Join X1 & X3:
X1[1]>X3[1])

After the update:

(i) Y = Y ∪ I

Y		
4	A	C
5	M	D
40	A	C
8	T	D
3	L	S

(ii) I = I: select I[1]<20
X1 is not updated because it is implicit

I		
4	A	C
5	M	D

(iii) I = Project([1],[3],[5])(Join I & X3: I[1]>X3[1])

I		
4	C	Q
5	D	Q

X5 = X5 ∪ I

X5		
4	C	Q
5	D	Q
8	D	Q
3	S	Q

Example 4 (Deletion)

X1			X3			X5		
8	T	D	8	T	D	2	Q	R
3	L	S	2	Q	R			
			3	L	R			

Relation Y as in Example 1

Delete from relation Y the tuples for which Y[1]>3.

This is equivalent to specifying the tuples to be deleted as

D = Y: select Y[1]>3

Relation D (the tuples to be deleted)

40	A	C
8	T	D

Definitions:

X1 = Y: select Y[1]<20
X5 = X3 difference X1

After the update:

(i) Y = Y difference D

Y		
3	L	S

(ii) D = D: select D[1]<20

D		
8	T	D

X1 = X1 difference D

X1		
3	L	S

(iii) The definition contains a difference and the second relation is to be updated.

D = X3 ∩ D

D		
8	T	D

X5 = X5 Union D

X5		
2	Q	R
8	T	D

The update continues as an insertion operation

Example 5 (Change in object value)

X1			X3			X5		
8	T	D	8	T	D	2	Q	R
3	L	S	2	Q	R			

Relation Y as in Example 1.

In relation Y replace the values Y[1]>3 by 10.

This is equivalent to the deletion operation of Example 4:

D

40	A	C
8	T	D

followed by the insertion

I

10	A	C
10	T	D

Definitions: X1 = Y: select Y[1]<20

X5 = X3 difference X1

After the update:

(i) Y = (Y difference D) union I

Y

3	L	S
10	A	C
10	T	D

(ii) D = D: select D[1]<20

D

8	T	D

I = I: select I[1]<20

I

10	A	C
10	T	D

X1 = (X1 difference D) union I

X1

3	L	S
10	A	C
10	T	D

(iii) D = X3 intersection D

D

8	T	D

(I = I)

I NULL

X5 = (X5 union D) difference I

X5

2	Q	R
8	T	D

(i) No operation is performed on the updating relation.
[Alternatively, the same result will be reached if the definition is applied to the updating relation with the difference replaced by intersection.]

The updating relation is passed to the lower levels as a deletion and the process continues as a deletion operation.

(ii) If the relation to be updated is explicit, the updated form will be the difference of the relation and the updating relation [resulting in (i)]. This applies to all the relations at lower levels.

Deletions

In this case the update may be specified by providing either a relation (D) containing the tuples to be deleted or a boolean filter which selects the tuples to be deleted from the relation to be updated. The extracted tuples constitute the updating relation (D).

Example 5 illustrates the mechanism of the deletion. It is the same as the mechanism of the insertion (described above) except for the following:

In (a) and (b) the union is replaced by the difference.

In (i) only the alternative method is applicable. The updating relation is passed to lower levels as an insertion.

In (ii) the difference is replaced by the union.

Change in object value

This is explained in Example 5.

The algorithm

The insertion algorithm for updates from higher levels is as follows:

```
procedure insert (R,I,DIFF2) recursive;
        boolean DIFF2; relations R,I,IN;
    comment  R      is the relation to be updated
             I      is the relation of tuples to be inserted
                      in R
             n      the number of relations defined on R
             D_k    is the k^th definition from R to the
                      lower level 1≤k≤n
             D*_k   is D_k with difference replaced by
                      intersection applicable in the case
                      of deletion
             DIFF2  a logical variable true when D_k contains
                      difference and the second relation is
                      to be updated
             X_k    is the k^th relation defined on R;
        if R is implicit then goto EXP;
      if DIFF2 then R:=R-I else R:=R ∪I;
    EXP: if n=0 then goto EXIT;
         for j=1 step 1 until n do;
       begin; DIFF2=DIFF2 and <D_j contains difference and
                the j^th relation defined on R is the second
                relation>;
              if DIFF2 then IN=I;
```

```
                    comment in case of deletion IN:=D*j(I);
                else IN:=Dj(I);
            if IN¬=null then
                insert(Xj,IN,DIFF2);
        end;
   EXIT:
end insert;
```

The algorithms for deletion and change of object value follow from the above algorithm.

Discussion

The significant advantage of this approach to updating relations is the restriction of updates to explicit forms only, which eliminates the potential cost of creating all relations down the hierarchy. One main disadvantage, however, is that the application programmer has to give the system an update routine for each function he adds to the system.

For example:

The definition: X1=SORT(Y) ON Domain(2);

When the updating relation I is inserted

Isorted=SORT(I) ON Domain(2);

X1=X1 Union Isorted;

This is a wrong result. The correct result is

X1=X1 MERGE Isorted

(The only change is the replacement of union by MERGE, which is a restricted form of the union.)

In the case of the deletion:

After deletion Y=Y difference D

Dsorted=SORT(D) on Domain(2)

X1=X1 difference D

or more efficiently

X1=X1 DEMERGE Dsorted

where DEMERGE is an operator which takes a tuple from the second relation and searches the first relation, tuple by tuple, until it finds

either a tuple equivalent to that of the second relation, in which case the tuple of the first relation will be deleted;

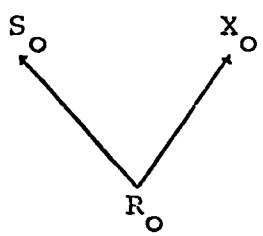
or a tuple of sort key (sort domain(s)) value greater than that of the second relation's tuple.

It repeats this until one of the two relations is exhausted.

The above update mechanism will be extended at implementation stage by adding rules which account for complex cases, e.g. when the defining relation appears more than once in a single operation.

e.g. R_o is defined as S_o join S_o ($R_o=S_o*S_o$)

Let $R_o=S_o*X_o$ (where $X_o=S_o$, X_o is a dummy relation)



To update S_o and X_o by I.

Since S_o and X_o are at the same level we can start by updating any branch to the next level.

Starting from left to right:

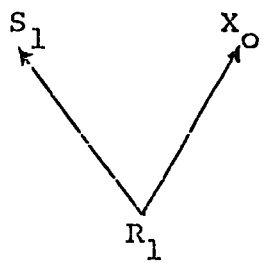
1. The relation to be updated is S_o

   (a) Update S_o

   $S_1 = S_o \cup I$ ($S_1, R_1, \ldots$ are the updated versions of $S_o, R_o, \ldots$)

   (b) Update R_o

   $R_1 = R_o \cup (I * X_o)$



2. The relation to be updated is X_o

   (a) Update X_o

   $X_1 = X_o \cup I$ (a dummy update)

   (b) Update R_1

   $R_2 = R_1 \cup (S_1 * I)$

i.e. updated $R = R_o \cup (I * X_o) \cup ((S_o \cup I) * I)$

$= R_o \cup (I * S_o) \cup (S_o * I) \cup (I * I)$

Generally, when the defining relation occurs more than once in a single operation of the definition, occurrences are updated one at a time keeping the other occurrences fixed at their last value. The update for each occurrence is carried to the next level.

The proof of the algorithm for higher level update

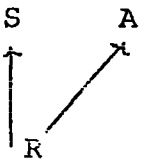
(1) I is the set of tuples to be inserted

S_o is relation S after I is inserted (after update)

R_o is relation R after update

The insertion is defined as:

$$S_o = S \cup I$$



Given (i) relation R is defined on relations S and A as $R = A \cap S$

(ii) the definition holds after R and S are updated, i.e. $R_o = A \cap S_o$

To prove that

$$R_o = R \cup I_o$$

where $I_o = A \cap I$ (the method followed in the insertion algorithm)

Proof

$$R_o = A \cap S_o$$

$$= A \cap (S \cup I)$$

$$= (A \cap S) \cup (A \cap I) \quad \text{identity}$$

$$= R \cap I_o$$

(2) The proof of the insertion algorithm at higher levels when the definition contains difference and the second relation is to be updated.

Given (i) $S_o = S \cup I$ the definition of insertion

(ii) $R = A - S$ the definition of relation R on relation S

(iii) $R_o = A - S_o$ the definition holds after the update

Prove $R_o = R - I$

Proof

$$R_o = A - S_o$$
$$= A \cap \overline{S_o}$$
$$= A \cap \overline{(S \cup I)}$$
$$= A \cap (\overline{S} \cap \overline{I})$$
$$= (A \cap \overline{S}) \cap \overline{I}$$
$$= R \cap \overline{I}$$
$$= \underline{R - I}$$

(2) The alternative approach for definitions containing difference and the second relation is to be updated.

Given $S_o = S \cup I$ the definition of insertion

$R = A - S$ the definition of relation R on relation S

$R_o = A - S_o$ the definition holds after the update

$I_o = A \cap I$ the insertion

Prove $R_o = R - I_o$

<u>Proof</u> (i) $R_o = A - S_o$

$= A \cap \overline{S_o}$

$= A \cap \overline{(S \cup I)}$

$= A \cap (\overline{S} \cap \overline{I})$

$= A \cap \overline{S} \cap \overline{I}$

(ii) $R - I_o = (A - S) - (A \cap I)$

$= (A \cap \overline{S}) \cap \overline{(A \cap I)}$

$= A \cap \overline{S} \cap (\overline{A} \cup \overline{I})$

$= \overline{S} \cap [A \cap (\overline{A} \cup \overline{I})]$

$= \overline{S} \cap [(A \cap \overline{A}) \cup (A \cap \overline{I})]$

$= \overline{S} \cap [\emptyset \cup (A \cap \overline{I})]$

$= \overline{S} \cap (A \cup \overline{I})$

$= A \cap \overline{S} \cap \overline{I}$

$= R_o$

In the case of deletions $S_o = S - I$ and $R_o = R \cup I_o$. The proof follows from the above proof.

Example 6 (Insertion at a lower level)

Definitions:

```
X1=Y:Y[1]>20
X5=project X1 on ([1],[4]);
```

Y

25	A	B	C
3	D	E	F

(a) To update X5 by relation I

I

30	C
40	F

(b) To update X1 by I

30	C	Q	R
40	M	N	P
10	S	E	M

3.1.2 UPDATE AT A LOWER LEVEL

In addition to the conditions imposed on ordinary updates, e.g. the compatibility of the updating relation and the relation to be updated, the following conditions should be satisfied for updating relations at lower levels:

(i) The update should not result in loss of information at upper levels, e.g. in Example 6(a) when relation X5 is updated, the updating tuples cannot be united with relation X1 because these tuples supply information for only two of the domains of X1. In such cases the update will provide incomplete information and should therefore be prohibited. Updating X1, however, does not lead to missing information in upper levels. The system should prompt the user to update the relation at the lowest level whose update does not violate conditions (i) and (ii).

(ii) No ambiguity should result at higher levels due to the update, e.g. relation R is defined as the union of relations S and Q. When R is updated the data base system cannot readily know which of the updating tuples should update each of S and Q and which should update both relations. Whenever such an ambiguity exists, the update operation must be prohibited.

(iii) The updating tuples must satisfy the definition of the relation to be updated. In Example 6(b), if

relation X1 is updated by relation I, the system must reject the last tuple because it contradicts the definition of the relation to which it will belong. Similarly, if a tuple is to be deleted from such a relation, an equivalent tuple must exist in the relation; otherwise the deletion is meaningless.

With these constraints, more weight is attached to the consistency of the data base information on the expense of the user convenience. Indeed, inconsistency in itself, regardless of any other repercussions, may perhaps lead to more inconvenience to the user.

With updating at a lower level, almost each operator in the definition requires a different updating algorithm. Some operators in the definition cause ambiguity with insertions only while they do not cause ambiguity with deletions. For other operators, the defining relations in the upper level must be explicit. Even if the update operation is to be delayed until the implicit relation is requested, the implicit relation has to be materialised in order to check the validity of the update. However, all the definitions do not lead to ambiguity in the case of deletions.

It would therefore be a sensible decision to prohibit insertions and value changes from lower levels. However, let us see the behaviour of some definitions containing the usual relational operators. The relational operators may be divided into two groups according to their performance when

relations having these operators in their definitions are updated.

(1) The regular operators

These operators have two major properties when the definition of the relation to be updated contains one of them:

(a) they do not lead to ambiguity

(b) the updating relation (tuple) can be passed to higher levels

A third property is not possessed by all these operators:

(c) they do not require the presence of the defining relation in its explicit form

In short, they almost have the same advantages of updates at higher levels. These operators are:

(i) Selection:

Definition X1=Y:Y[1]>2

When I is inserted in X1 Yupdated=Y $\cup$ I

The update to be passed to upper levels = I

When D is deleted from X1 Yupdated=Y-D

The update to be passed to upper levels = D

(ii) Difference:

Definition X1=Y-X

After insertion X1=X1 $\cup$ I

Y=Y $\cup$ I

X is not affected by the insertion

i.e. the first relation is updated by I.

I is passed to higher levels

Since $I \cap X = \emptyset$, the second relation should be made explicit to check the validity of the update.

After deletion X1=X1-D

Y=Y-D

X is not affected by the deletion

i.e. the first relation is updated by D.

D is passed to upper levels

(iii) Join:

Definition X1=Join(Y:Y[1]>2

& X:X[1]<5)

After insertion: $X1 = X1 \cup I$

$Y = Y \cup I_y$

$X = X \cup I_x$

where I_y= projection (on the domains of Y)(I)

I_x= projection (on the domains of X)(I)

Validity tests:

(1) tuples of I_x and I_y must satisfy their corresponding term of the boolean filter.

(2) either I_x must contain (X:X[1]<5)

or I_y must contain (Y:Y[1]>2)

The second condition necessitates the presence of the defining relations (X and Y) in their explicit forms. It is advisable to make one relation (e.g. X) explicit and if condition (2) is satisfied, the updating tuples of the other relation (e.g. Y) will be passed to higher levels without the need to make Y explicit.

After deletion: $Y=Y-D_y$

$X=X-D_x$

where D_x = projection (on the domains of X)(D)

D_y = projection (on the domains of Y)(D)

The above conditions for the validity of the insertion apply for the deletion.

(2) The irregular operators

When a definition contains one of these operators, the following problems arise:

(a) ambiguity of insertions.

(b) the defining relations have to be explicit in order to either check the validity of the update or to pass the tuples to be inserted to higher levels.

These operators are:

(i) The union

e.g. Definition (a) $X1=X \cup Y$

After insertion $X1=X1 \cup 1$

X and Y cannot be updated (by insertion)

It is sometimes possible to avoid ambiguity, for example consider the following definitions:

(b) $X1=Y:Y[1]>20$

(c) $X2=Y:Y[1]\leq 20$

(d) $X3=X1 \cup X2$

i.e. X1 and X2 are defined on one relation and are disjoint

projection([1])(X1) $\cap$ projection([1])(X2) = $\emptyset$

Now, let us consider the deletion using definition (a).

After deletion X1=X1-D

$$X=X-D_x$$

$$Y=Y-D_y$$

where $D_x = X \cap D$

$D_y = Y \cap D$

D_x and D_y will be passed to higher levels.

Since deletions are possible, changes in value will be possible because the tuples whose objects are to be changed have been known from the deletion operation. It should be noted, however, that for the deletions and value changes to take place, the defining relations must be explicit.

(ii) The projection

Definition X1=projection([1],[2])(Y)

After insertion $X1=X1 \cup I$

Y cannot be updated (by insertion).

After deletion X1=X1-D

$$Y=Y-D_y$$

where D_y = projection (on domains of Y)(Join Y and D)

(Y:Y[1]=D[1] & Y[2]=D[2]).

D_y is passed to higher levels.

Similarly, since deletion is possible, change in object values is also possible.

Alternatively:

Consider the following example:

Y				X1	
4	5	A	8	5	8
2	6	B	4	6	4
1	2	C	3	2	3
3	4	D	8	4	8
1	4	E	7		

X1 is defined as project ([2],[4])(Y)

Now let us delete D

2	3
4	8

After deletion X1

5	8
6	4

To update Y, we will delete all the tuples of Y whose second and fourth object values equal the first and the second values of any tuple of D.

After deletion Y

4	5	A	8
2	6	B	4
1	4	E	7

This can be generalised as follows:

Y after update = $Y-D_y$

D_y (tuples deleted from Y) =

$$\bigcup_{j=1}^{m} Y:(Y[i[1]]=D_j[1]\ \&\ Y[i[2]]=D_j[2]\ \&\ \ldots\ \&\ Y[i[p]=D_j[p]])$$

where i[i] is the i^{th} domain in the projection list,

e.g. in the above example the list is (2,4)

$D_j[r]$ is the r^{th} object (Domain (r)) of the j^{th} tuple of relation D

r=1,...,p } where p,m are the degree and

j=1,...,m } cardinality of relation D

Now, if Y is a defined relation and is implicit, we will pass D upwards after noting the domain of Y corresponding to each domain of D. This information is in the definition. The above expression will then be applied to relations in upper levels.

Within these limits the defining relation need not be explicit. However, if the defining relation (the relation to be updated, e.g. Y) has another relation defined on it and the definition of the latter relation contains selections or joins on domains other than those in the projection list, the defining relation (Y) must be explicit.

(iii) Some user-written operations

e.g. Suppose a user wants to keep a relation sorted on a certain domain. He will submit the following definition to the system:

X1=sort(Y) on domain (1)

After insertion X1=X1 MERGE(sort(I))

$Y=Y \cup I$

I is passed to higher levels.

After deletion X1=X1-D, or for better performance,

X1=X1 DEMERGE(sort(D))

Y=Y-D

D will be passed to higher levels.

This example shows the following:

(a) both insertions and deletions are possible.

(b) the tuples to be inserted must be sorted before

they are inserted (merged). The system should know how to deal with the updates at the time of the submission of the definition. This is a necessary condition for all definitions containing user-defined operators unknown to the system.

(c) the defining relation need not be explicit.

Local Insertions

A possible solution for the problem of insertion at lower levels when definitions contain the union or the projection operators follows.

In such a situation we cannot add the updating information to the data base because it leads to ambiguity. A possible improvement in the situation is that the system creates a base relation containing all the insertions.

e.g. relation C is defined C=A∪ B

When tuples are inserted in C, a base relation T is created by the system, i.e. Define C=A∪BUT.

Whenever tuples are inserted in C, they are kept in T.

The insertion is not carried to all the relations above C, It is retained locally and C has up-to-date information.

3.1.3 SUMMARY OF UPDATES

The above discussion shows that it is possible to update relations at higher levels without the need to recreate

implicit relations.

On updating from lower levels, the union and the projection operations require more control information from the user in order to achieve successful insertions. Deletions and changes in values are possible.

With user-written functions, the operation to be carried out on updating relations should be specified.

Having dealt with the update problem, let us discuss the problems associated with the definition and deletion of relations. Since these are system operators the rules defining their operation may change with changing systems or implementations. Here one way of solving the problem is suggested, bearing in in mind the objectives set out at the beginning of the chapter.

3.2 DELETION OF RELATIONS

By deletion of a relation is meant, in the user's view, the total removal of the relation: its name and information content from the data base, and the freeing of the space it occupies.

The following rule is suggested:

> All relations may be deleted by authorised users except defining base relations.

This may be explained as follows:

(a) In order to delete a defining base relation (e.g. X and Y in Figure 2.1) the user must first delete the relations dependent on it. However, if he is not authorised to access some of the dependent relations, he will not be

aware of their presence and he will therefore delete his own dependent relations.

(b) A defined relation at the bottom of the hierarchy may be deleted (e.g. relations Y1 and X4).

(c) A relation in the middle of the hierarchy (i.e. a defined defining relation, e.g. X1, X2 and X3) may be deleted. The system deletes the physical representation of the relation but it keeps the definition. It then assigns the definition to a dummy name. The dummy name is substituted in the definition of the dependent successors,

e.g. Definitions X1=Y:Y[1]>5

X5=X1:X1[2]>10

X6=X1:X1[2]≤10

Delete (X1) changes the above definitions as follows:

dummy=Y:Y[1]>5

X5=dummy:dummy[2]>10

X6=dummy:dummy[2]≤10

Dummy is only a definition to pass updates. It will not be physically realised.

The Deletion Algorithm

Delete (R) recursive.

A. If R is a defined relation goto B;

(R is a base relation)

A1: if R is a defining relation goto A3;

A2: (Base non-defining relation)

Destroy relation R; goto FINISH;

A3: (Base defining relation)

If the user is authorised to delete some of the relations defined on R then goto A5;

A4: (user unauthorised).

Print message 'You are not authorised to delete R'.

goto FINISH.

A5: (user authorised to delete some or all the relations dependent on R)

List the relations dependent on R which the user is allowed to delete.

Ask the user if he wants all these relations deleted?

If the answer is Yes goto A7;

A6: (Answer is NO)

Print message 'R cannot be deleted. You may delete some of these relations'.

goto FINISH.

A7: If these are all the relations dependent on R goto A9;

A8: (some other relations are defined on R)

Destroy the user's relations dependent on R. List all the relations that have just been destroyed. Print 'R has been destroyed'.

Mark R not authorised to the current user.

goto FINISH.

A9: (the user is authorised to delete all relations dependent on R)

Destroy R and dependent relations. List the destroyed relations.

goto FINISH.

B. (R is a defined relation)

B1: Is any relation dependent on R? If Yes goto C.

B2: (R is the lowest in the hierarchy)

Destroy R.

B3: Is the new lowest relation a dummy definition? If NO goto FINISH.

Remove the dummy definition.

goto B3.

C. (R is a defined relation with some relations dependent on it)

Destroy R (if physically present)

Assign the definition to a dummy relation name.

Substitute the dummy name in the definitions of dependent relations.

FINISH:

3.3 THE DEFINITION OF RELATIONS ON OTHER RELATIONS

3.3.1 CIRCULAR DEFINITIONS

This is the definition of a relation directly or indirectly in terms of itself.

With circular definitions the system can store the same data in more than one form. Depending on the usage and the other characteristics of the data, the system can choose which forms to store such that the storage utilisation is optimum.

e.g. Consider the following definitions:

(a) $Y = X1 \cup Y1$

(b) $\begin{cases} X1 = Y : Y[1] > 20 \\ Y1 = Y : Y[1] \leq 20 \end{cases}$

The information in relation Y may be stored in one of two forms:

(a) as two relations X1 and Y1 while Y is now a defined relation to be assembled only when it is requested.

(b) as one relation Y with X1 and Y1 as defined relations.

However, serious inconsistency may arise if this facility is not carefully used.

In the previous example X1 and Y1 are disjoint and the definitions are always true. Now let us consider the following example:

Define 1. $Y1 = Y : Y[1] > 20$

2. $X1 = Y : Y[2] > 10$

3. $\underline{Y = Y1 \cup X1}$

It is not always obvious if these definitions are consistent. The system has to perform much time consuming testing before it decides to accept or reject the third definition. Even if the definitions are consistent, a consistent update for X1 in (2) will not necessarily be consistent for (3).

As definitions become more complex the overhead for checking the validity of updates and definitions becomes enormous. It is, therefore, sensible not to allow the user to submit such definitions.

However, it is suggested to allow the system programmer to submit definitions of the following type:

Define Y1=Y:Y[1]>20 } major path
X1=Y:Y[1]≤20 }

Define (auxiliary path) Y=X1 ∪ Y1

The system uses the major definition only. At reorganisation time it decides which path gives better performance. The preferred path is made a major path and its definition is used. The other definition becomes an auxiliary path.

This concept can be extended such that the users can prompt the system on possible auxiliary paths. Because the check for consistency occurs only at reorganisation time (or after a reasonably long period of time) the overhead will be within a tolerable limit. In this way the inconsistency can be avoided at a low cost while benefiting from the main advantage of circular definitions.

3.3.2 REDEFINITION OF RELATIONS

e.g. If relation X1 is defined as X1=Y:Y[1]>20
and relation X5 is dependent on X1;
then if X1 is redefined as X1=Y:Y[1]<20
the information in X5 will be inconsistent with
its definition.

It is required to set some rules to avoid such inconsistency. However, if no relation is dependent on the redefined relation, then the redefinition is harmless.

It is suggested that redefinition is to be allowed and that the system seeks methods by means of which it can avoid inconsistency. One method is suggested below.

The redefinition of a relation should be seen to amount to the deletion of the relation and the definition of another relation having the same name as the deleted one.

If the user intends to redefine relation R then he is

either aware of the relations dependent on R and he may or may not want their information changed in accordance with the new definition;

or he is not aware of the dependent relations (he may not be authorised to access the relations dependent on R).

The following course of action is suggested to be followed by the system:

A. If R is a base relation then it cannot be redefined. Goto FINISH;

B. If no relation depends on R then R will be redefined and the old data is destroyed. Goto FINISH;

C. If the user is not authorised to delete some of the relations dependent on R then goto G.

D. The system gives a list of the dependent relations authorised to the user and asks the user if he intends to have these relations affected by the new definition. If he does not want them affected goto G.

E. Make R and all his relations that depend on R implicit.

F. Redefine R. Goto FINISH;

G. Change the definition of R replacing R by a dummy name. Substitute the dummy name in his relations dependent on R (if any). Delete the physical representation of R.

H. Enter R as a new relation defined on its predecessors.

FINISH:

By this method the user is free to redefine his relations without creating inconsistencies in the information or the definitions of relations.

3.3.3 ASSIGNMENT OF DEFINED RELATIONS TO RELATIONS

(i) When a relation is assigned to a defined relation the resulting relation carries the information of the defined relation at the time of the assignment. The resulting relation is independent of the hierarchy and has no connection with what happens to the defined relation.

(ii) If a defined relation is assigned to another relation, the new information in the defined relation may not be consistent with the definition.

The following method is suggested:
For the instruction R:=S (meaning set the content of relation R equal to the content of relation S) where R is a defined relation:

A. It is advisable to first check that the user wants R assigned to S. A wrong instruction may cost time

consuming operations.

B. The assignment is considered as a deletion of the tuples of R followed by the insertion of S in R. Before doing that, the validity of S to update R is checked, i.e. the new information in R should be consistent with the definitions at higher levels (as explained in the previous section).

C. Make the relations dependent on R implicit and assign and copy S to R.

3.4 FINAL REMARK

(i) Some of the problems considered above apply to relational and non-relational data bases (e.g. the question of how far the user is allowed to change the subschema arises in non-relational data bases). The relational model with its underlying well-defined mathematical operations makes the approach systematic and possible to generalise, as has been shown in the case of the update.

(ii) For the purpose of illustration the author has implemented some of his above-mentioned recommendations in IS/1.0. The listing of an IS/1.0 session is shown in Figure 2.2.

```
ISX402I IS/1 SESSION START-UP, DATE=73.04.03, TIME=16.58
ISX452R COMMAND INPUT ASSIGNED TO READER
ISX452R MESSAGE OUTPUT ASSIGNED TO PRINTER
ISX452R RESULTS ASSIGNED TO TERMINAL
ISX452R ***** ASSIGNED TO NOLIST
ISX450I MESSAGE LEVEL LOGIN LOGCLT;
ISX450I ENTOPT
ISX431C PLEASE SIGNON
ISX450I SIGNON(IZZ);
ISX433R THANK YOU, IZZ - PLEASE GO ON
ISX450I ASSIGN RESULTS PRINTER;
ISX452R RESULTS ASSIGNED TO PRINTER
ISX450I  /* DCL DOES THE FUNCTION OF DEFINE*/
ISX450I DCL(X2:LOAD(X),SELECT(TCP(1)=CANADA),LOAD LL('NORTH A'),CC);
X2          IS DEFINED ON X
ISX101I CHANGE CONFIRMED AT 16.59.57
ISX450I  /*REORDER IS EQUIVALENT TO PROJECT*/
ISX450I DCL(X4:LOAD(X2),REORDER(2,3));
X4          IS DEFINED ON X2
ISX450I CHANGE CONFIRMED AT 17.00.12
ISX450I  DCL(X6:LOAD(X4), LOAD(Y), JOIN(TCP(1)=PEN(1)));
X6          IS DEFINED ON X4          AND Y
ISX450I CHANGE CONFIRMED AT 17.00.19
ISX450I /* DATA HAS ALREADY BEEN LOADED IN RELATION I*/
ISX450I INSERT(I,X4);
ISX450I /*INSERT TAKES 1ST RELATION AND INSERTS ITS TUPLES IN 2ND RELN*/
X4          CAN  NOT BE UPDATED .PLEASE  TRY TO UPDATE RELATION
X2
ISX450I  SCRUB(Y);
ISX450I  /*SCRUB IS EQUIVALENT TO DELETE*/
YOU CAN NOT DELETE Y        .YOU MAY DELETE THE RELATIONS DEFINED ON IT:
X6
X4
X2
ISX450I SCRUB(X2);
X2  HAS  BEEN DELETED
ISX450I CHANGE CONFIRMED AT 17.00.56
ISX450I DCL(X4:LOAD(S),LOAD(R),SELECT(TCP(1)=PEN(1)));
ISX450I /*REDEFINITION OF X4*/
THE FOLLOWING RELATIONS ARE DEFINED ON X4        :
X6
DO YOU WANT TO REDEFINE X4          ?
YES;/*USER'S RESPONSE*/
X4          IS DEFINED ON S           AND R
ISX450I CHANGE CONFIRMED AT 17.01.09
ISX450I END; YES;
ISX436C IS/1 SESSION SHUT-DOWN IMMINENT - PLEASE CONFIRM
ISX101I CHANGE CONFIRMED AT 17.01.20
ISX427C END OF IS/1 SESSION, TIME=17.00
```

Chapter 3

A GENERALISED PAGE REPLACEMENT ALGORITHM

1. Introduction

In a relational data base having a defined relations facility, some relations are maintained in definition form until queried while others are explicit. An efficient replacement algorithm is needed to manage the content of the free space available for the data base as defined relations alternate between the state of definition and the state of explicit representation. Such an environment is analogous to a virtual memory environment and the defined relation is analogous to a page.

In this chapter a replacement algorithm for defined relations is described. It is discussed as a generalised page replacement problem in which the pages have variable sizes and the cost of a page fault is a function of the particular page referenced. It is shown how the conventional page replacement algorithms are found to perform inadequately.

New algorithms are proposed for reducing the cost incurred because of page faults in response to a series of references. Also included are the results of simulation experiments which have been run in order to compare the cost of performance of these algorithms with standard techniques and with the minimum avhievable cost.

The problem of dependencies existing among relations (i.e. relations defined on other defined relations) is left for the next chapter where the replacement algorithms are adjusted to deal with the problem.

2. The Statement of the Problem

The total data storage space available to the relational system may be visualised as consisting of two parts (see Figure 3.1):

(1) a base storage area in which is stored the relations from which all defined relations are ultimately computed, and

(2) a dynamic storage area (or the free space in the data base) in which the system may temporarily maintain explicit forms.

Data removed from the dynamic store is not lost since it can always be reconstituted from base data by applying the appropriate definition. It is assumed that a pre-assigned amount of space is available from dynamic storage, and that the system governs the contents of this area. The problem of managing the dynamic area so as to maximise overall efficiency is analysed.

At some stage in the process of creating and querying relations, one or more explicit relations will have to be deleted from the dynamic storage in order to free space for other requested relations. A "replacement algorithm" is the name for the process that decides which relation is to be overwritten. Ideally the replacement algorithm should function so as to minimize some overall measure of the cost of meeting the storage constraint in responding to a series of queries.

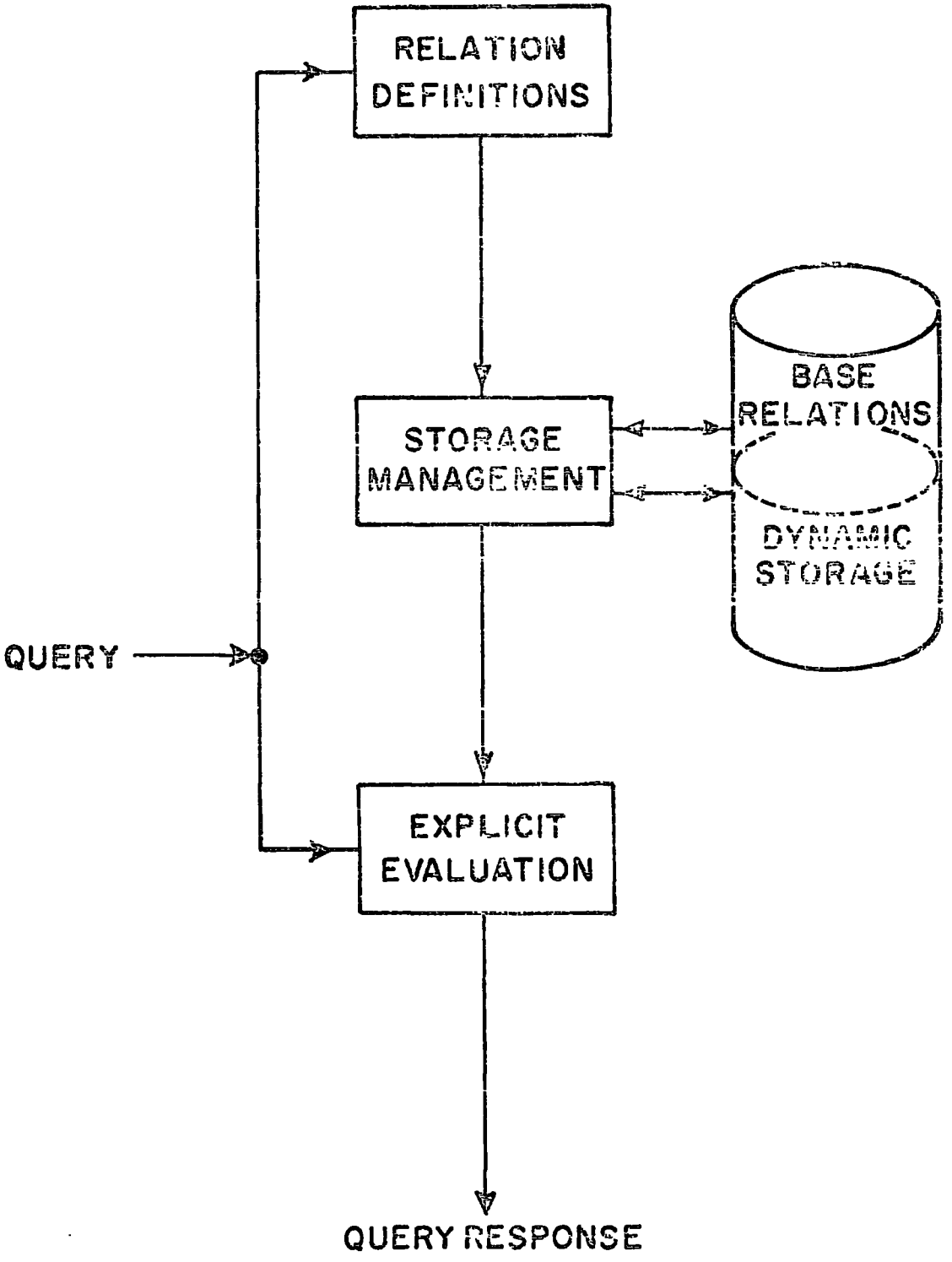


FIGURE 3.1 Base and dynamic storage areas

It is noteworthy that this problem is a variation, indeed an extension, of the conventional demand paging problem [Mattson et al 1970, McKeay and Hoare 1972]. In a memory hierarchy, core storage space typically is less than the sum of data requested. Pages of data must be traded back and forth between a bulk storage device (e.g. disk) and the high-speed core. Rules must be put into effect for determining which blocks of data should be maintained in core and which should be replaced by freshly referenced data. The system tries to estimate which of the pages currently in core will be referenced in the near future, and which pages will not be needed for a long time.

Ordinarily the hierarchy implementation is such that both the page size and the cost of fetching a page from bulk storage are constant. In contrast to this, the cost of creating a defined relation depends on the complexity of its definition, and relations so formed will vary in size. Thus the problem of anticipating queries in order most effectively to assign relations to either implicit or explicit forms is equivalent to that of designing a replacement algorithm for a memory hierarchy employing unequal page sizes and variable fetch costs. Table 3.2 illustrates the correspondence in detail.

3. Replacement Techniques

In this section the performance of conventional paging algorithms in the generalised paging environment is examined, and two new replacement techniques are proposed. These methods are compared against idealised replacement strategies.

CONVENTIONAL PAGING	IMPLIED RELATIONS
Page exception, i.e. page needed is not in core	Required relation is in implicit form
ACTION:	
1) Select a page to be replaced	1) Select one or more explicit relations to be dropped
2) Page in the required page	2) Convert the required relation from implicit to explicit
COST:	
Cost is due to reading in a page	Cost is due to computing the implicit relation
Cost is usually fixed	Cost varies from one relation to another. The cost for one relation is time varying
Overhead of the replacement algorithm is significant	Overhead cost not as critical
PAGE SIZE:	
Page size is usually fixed	The size varies considerably from one relation to another. The size of one relationis time varying

Table 3.2

The effects of updates are not directly considered in this analysis. In order to consider updates the model assumes the particular strategy explained in the previous chapter, i.e. relations are updated in place and that an implicit relation is not made explicit for updating. Updates are accounted for by permitting cost and storage parameters to be time varying.

In view of the dual nature of the replacement problem considered here, the term "page" will often be used to refer to a relation in its role as an object to be shuffled in and out of a limited storage space. For simplicity the algorithms are described as if only a single page needs to be replaced. This is strictly true only if the page sizes are uniform. In the case where the incoming page requires additional space the algorithms must be applied iteratively, eliminating successive pages until sufficient storage is free.

In the discussion, reference is made to time, which is thought of as a discrete sequence indicator t = 1,2,3...

The observed number of references to the i^{th} relation up to time t is denoted u_i^t (i = 1,2,...,r where r is the number of defined relations). The processing cost to create the relation from its definition at time t is denoted c_i^t, and the storage space required is identified by s_i^t.

The performance of the following replacement algorithms is compared in a relational context:

1) Least Recently Used

2) Least Frequently Used

3) Least Expected Cost

4) Least Expected Loss per unit size

5) Arbitrary Replacement

1) THE LEAST RECENTLY USED (LRU)

The page selected for replacement is the one that has not been referenced for the longest time [Mattson et al 1970]. As t = o we define variables $w_i^o=o$ for each i=1,2,...,n. At t = k when a reference is made to page i

$$w_i^k=k \text{ and } w_j^k=w_j^{k-1} \text{ for } j \neq i.$$

The LRU rule is: at time t replace the page i for which

$$w_i^t<w_j^t \quad j=1,2,\ldots,n \quad j \neq i$$

This rule is frequently cited in studies of conventional paging systems. Note, however, that it does not employ size and cost characteristics in arriving at a replacement decision. Therefore it is to be expected that LRU is most appropriate for the case of uniform size and cost, and will not extend readily to the more general case.

2) LEAST FREQUENTLY USED (LFU)

Under LFU the page replaced from the storage at time t is that page that has been referenced the fewest times over the interval 1,2,3,...,t. If the least frequency of usage is possessed by two or more pages an arbitrary rule is used to break the tie (e.g. the one that has a smaller reference number is replaced).

A precise description is as follows. Whenever a reference is made to page i at time t its reference count (usage count) is updated by 1.

$$u_i^t = u_i^{t-1} + 1 \quad i = 1...r$$

At any point t=k when the dynamic area is full and the need for replacement arises, replace page i which has the smallest reference count, i.e.

$$u_i^k < u_j^k \quad j = 1,...r, \quad j \neq i$$

This algorithm has the advantage of simplicity in application and small overhead in storage of the data required for the algorithm.

3) LEAST EXPECTED COST (LEC)

The expected cost of fulfilling the next request if relation i is not explicit is the product of the probability of reference to i and the cost c_i^t.

At time t

$u_i^t = u^{t-1}_i + 1$ The reference count for page i.

The proportion of requests referencing page i up to time t is:

$$\frac{u_i^t}{t}$$

The expected loss when i is not in the storage may be estimated as:

$$\frac{u_i^t}{t} \times c_i^t$$

Hence the problem is to find the page with the least expected cost at time t.

i.e. $$\min_k \frac{u_k^t \times c_k^t}{t} \quad \ldots$$

Since t is constant in this ensemble, it suffices to replace that page i for which

$$(u_i^t \times c_i^t)$$

is least.

Note that if pages have equal cost then LEC is LFU.

4) LEAST EXPECTED LOSS PER UNIT SIZE (LECS)

This algorithm weights the expected cost used in (3) by the size of the page.

At time t replace page i that minimizes $\frac{u_i^t . c_i^t}{s_i^t}$ i=1,...r

This algorithm reduces the LFU when the cost and storage parameters (c_i^t, s_i^t) do not vary with the index i. In the variable parameter case it has the virtue that frequency of usage, creation cost and page size are all included in the replacement mechanism. The weighting is such that if two pages have identical cost and usage frequency, then the one occupying more space will be replaced. Such an argument supplies a heuristic justification for an inverse weighting of the page size.

This argument is further strengthened if one considers situations such as the following. Suppose that n pages of unit size and cost and one page having size=cost=n units are in the buffer at a given time t. Assume that all have equal usage, u_i^t. Which should be replaced by an incoming request for n units of storage?

Now the contribution to the expected cost at time t+1, if the large page is replaced (assuming that u_i^t are true probabilities), is $(u_i^t)(n)$. If the small pages are all replaced, the expected cost due to one of them being requested at time t+1 is $(nu_i^t).1$. Therefore the loss is the same in either case, and the LEI criterion reflects this.

In a later section, the solution of an analytic model of the replacement problem in order to obtain the so-called "preferred set" will provide a more formal justification of the LECS method.

5) ARBITRARY REPLACEMENT (ARB)

This algorithm replaces a randomly chosen page. It is useful in order to demonstrate the actual improvement in performance obtained by each algorithm over arbitrary selection. An algorithm whose performance is the same as or worse than ARB may reasonably be discarded.

At time t=k choose the page to be replaced as follows:

(i) generate a random number m between 1 and n (where n is the number of pages in the

dynamic area).

(ii) replace the m^{th} page in the buffer.

4. Ideal Replacement

This replacement algorithm is suggested and proved by Casey and programmed by the author [Casey and Osman, April 1974]

The performance indicator of a page replacement algorithm has been taken to be the sum of the costs of reconstituting individual relations in response to a given sequence of requests. Replacement algorithms up to this point have been assumed to be non-anticipatory; they receive no information regarding an incoming request until the moment of decision, when storage must be provided for the referenced data.

In order to evaluate these algorithms, it is pertinent to inquire what the minimum achievable cost might be for a given reference string. To ask this question is to seek the performance of an algorithm that examines the whole request sequence in advance, and formulates an optimal series of replacements using this information. This algorithm "cheats" in that it employs information not available to the other routines. Consequently, it is not a competing technique for fulfilling requests upon demand. On the other hand, the performance it attains is a feasible upper limit in the sense that a very "lucky" replacement algorithm could yield the same result in a given trial.

For the case of uniform page sizes and costs, several methods

exist for finding the ideal performance [Belady 1966, Mattson et al 1970, Parmelee et al 1972]. The optimal replacement strategy is not unique, so that different constructions are possible. Mattson et al have shown that one optimal rule (the so-called OPT algorithm) always replaces, among the pages currently in the 'buffer', that page which is requested furthest in the future. They proposed an algorithm for determining this replacement series in two passes over the reference string. Later work [Belady and Palermo 1974] has shown that a single pass will suffice.

These results do not appear to extend directly to systems possessing unequal storage sizes and costs. One can appreciate that an optimum rule must, roughly speaking, be reluctant to dismiss from the dynamic area an expensive relation that occupies little space. However, the precise method embodying such rules is a complicated function of the reference string and the storage and cost parameters. Unlike the OPT algorithm, it is not easily expressed in words.

The algorithm investigated here uses a search technique to determine an optimal replacement sequence. The method is best illustrated by a graphical construction, Appendix 4, figure A4. This construction shows the tree generated as various replacement decisions are examined in response to requests for new relations.

A node of the tree is associated with a particular list of storage contents. Such a list defines a "state" of the storage. Modifying the buffer in order to accommodate a request results in a new state. Ordinarily there are a number of choices for relations to be deleted to make way for referenced data. Thus from a given state there are a number of possible transitions to succeeding states. These transitions are indicated by the branches of Figure A4 (Appendix 4).

If all possible transitions are evaluated in response to a given reference string, then the tree generated grows exponentially as a function of the time index. Each path from root to leaves corresponds to a different replacement strategy. Somewhere in this exhaustive set of strategies are the minimum cost paths that we seek. However, exhaustive evaluation (i.e. generating the entire tree) is too expensive and time consuming for even moderate sized problems. The evaluation algorithm investigated here "prunes" the tree, eliminating states from further consideration while retaining an optimal path. In experiments this pruning has been found to be quite effective so that only a small number of states in addition to the optimal one survive each pruning step.

The algorithm may be described as follows. Consider the nodes remaining unpruned after the k^{th} request has been treated. To each node corresponds not only a state but a cost-to-date as well. From this set of survivors generate

the complete set of states attainable in response to the (k+1) request. Now prune these states using the following rule:

Pruning Rule

Definitions:

X_i = set of stored relations corresponding to state i

C_i = the cost incurred in arriving at state i from start

c_p = the cost of retrieving the p^{th} relation when it is not in the buffer

Designate by P_{ji} the set of relations that are in state j but not in state i. (In ordinary set theory parlance we would write $P_{jk} = X_j \sim X_i$). Then we prune state j if there exists another state, i, such that:

$$C_j \geq C_i + \sum_{p \epsilon P_{ji}} c(p)$$

This condition has a verbal interpretation. If the cost of arriving at state i added to the cost of transforming from state i to state j by retrieving the relations in P_{ji} is no greater than the cost associated with state j, then node j is pruned. A mathematical proof that this pruning algorithm retains an optimal path in the tree is given by Casey in [Casey and Osman, April 1974]. In Figure A4 the states pruned by the algorithm are shaded.

With the aid of pruning it has been practical to conduct evaluation experiments assuming hundreds of requests and 10-15 relations. On the other hand, the technique has limitations and would benefit from further refinements. To illustrate, suppose that in a particular case there are 20 relations of unit size and one relation of size 10 units, all sharing an area of size 20 units. If the buffer is full of unit-storage relations, then whenever the large relation is requested there are $\binom{20}{10} = 185{,}000$ possible transitions to new states. Furthermore, the pruning rule will not eliminate any of these nodes.

Attempts to improve the algorithm will centre on the use of future references to prune the tree more heavily. This would conceivably result in a rule analogous to that embodied in the OPT algorithm, except that the time until a relation is referenced must be weighted by its costs and size in determining whether it should be discarded.

5. Experiments

5.1 Measurement of the Relation Parameters

An experimental comparison of the page replacement algorithms was conducted using size and cost parameters of relations from the geological data base.

Some of the recurring queries were converted into 27 defined relations. The cpu time to create each relation from its definition was measured. Because the IS/1.0 system was cpu bound, the cpu time was a sufficient representation of the cost. The size was measured by counting the number of blocks

occupied by the explicit relation.

The most frequently used 11 relations were included in the comparisons. The characteristics of these eleven relations are listed in the following table.

PAGE (relation)	SIZE	COST
1	12	31
2	5	20
3	3	8
4	2	24
5	6	4
6	4	2
7	1	2
8	8	29
9	30	50
10	13	29
11	7	24

Cost and Size Parameters (11 relations). Size units are in 1kb blocks. The cost is scaled from measurements of cpu time required to create the relation.

In addition, the statistical properties of the 27 defined relations were investigated. No single standard probability curve closely matched the distribution of size and cpu cost associated with these relations; however, a normal density function gives more satisfactory agreement than a uniform or skewed distribution.

In the early version of the IS/1.0 system simple i/o

operations required an excessive amount of cpu activity, thus the correlation measured between size and cost was rather higher than one might expect in a commercial data base system. Even so the correlation coefficient was only 0.41. Another set of cost and size parameters was drawn from a data base concerned with land usage in the Greater London Council [Aldred et al 1974]. The correlation coefficient of size and cost is 0.35. For this application, also the size and cost parameters are not heavily correlated.

On the basis of this examination of data base characteristics it was decided to neglect correlation between size and cost, and to simulate larger data bases using size and cost parameters drawn independently from Gaussian populations having the observed sample means and standard deviations. Consequently, parameters were generated for a 100-relation collection having mean size equal to 36 units with a standard deviation of 37.2, and a mean cost of 120 with an s.d. of 137.6.

The tests were also repeated with independent parameters drawn from exponential, and from composite normal-exponential distributions, with results similar in nature to those that follow.

5.2 The Generation of Reference Strings

In order to test the algorithms, strings of random references (numbers) having several lengths and a variety of properties were constructed. Since the characteristics of the relation that affect the performance of replacement algorithms are the size, the

creation cost and the frequency of reference, strings having various levels of these properties were generated. Because of its slower processing speed the IDEAL algorithm was run only on the eleven relation set and with a string of length 500.

The reference strings were created as follows:

a) A reference string for the 11-relation data base. The string length is 500.

   i) Uniform distributions: each of the eleven relations had the same probability of occurrence.

   ii) Low cost weighting: the four low cost pages occurred three times as frequently as the remaining pages.

   iii) Large size weighting: the four pages with the largest size occurred three times as frequently as the remaining pages.

   iv) High cost weighting: the converse of (ii).

   v) Low size weighting: the converse of (iii).

b) A reference string for the 100-relation data base. The string length is 3000.

   vi) Low cost weighting: the twelve low cost pages were assigned five times the frequency of the remaining pages.

   vii) Large size weighting: the twelve small pages were assigned five times the frequency of the remaining pages.

viii) High cost weighting: the converse of (vi).

ix) Low size weighting: the converse of (vii).

In these strings, successive references were statistically independent, whereas in a real data base operation the sequence of references would probably be correlated. This condition would tend to improve the probability of reference to recently accessed relations and thus the replacement algorithms would perform better than in the simulation experiments. The random reference sequences employed here may be considered a "worst case" situation, useful for comparing techniques.

5.2.1 Generation of weighted strings of requests

To generate a reference string, s, of length L such that the relations having large sizes occur 'a' times as frequently as the remaining relations:

The total number of relations is m.
The total number of large size relations is n.
Let the large size relations be relations number w_1, w_2, w_3, ... and w_n.

1) j = o.
2) j = j+1. If j > L then STOP.
3) generate a random number, i, between 1 and m+(a-1)n.
4) if i > m go to (6).
5) s_j = i. go to (2).
6) k = entier ((i-m) / (a-1)).
7) $s_j = w_{k+1}$

go to (2).

5.3 The simulation program

The simulation program works as follows:

i) Input the relation parameters (sizes and creation costs), the cost of answering a query from explicit form, the dynamic storage sizes for which the experiment is run, the cost of deleting the explicit form of a relation and the reference string.

FOR EACH STORAGE SIZE REPEAT STEPS (ii) TO (xiii)

ii) Set total cost, the number of successes (hits) and the number of relation swaps = o. Set the reference counts of all the relations = o.

iii) Pick a reference from the string, reference to relation I.

iv) Update the reference count of relation I.

v) If relation I is EXPLICIT then go to (x).

vi) Update the number of page faults.

vii) If the available storage > the size of I go to (xi).

viii) Using the replacement algorithm evaluate the replacement criterion for all the explicit relations and choose the relation, k, having the minimum value of the criterion.

ix) Add the size of relation k to the size of the available storage.
Set relation k implicit.

Update the number of relation swaps. Update the total cost by the cost of a deletion. Go to (vii).

x) (Satisfy the query from the explicit form.)
Update the total cost by the cost of accessing an explicit form.
Update the number of successes(hits). Go to (xii).

xi) Create Relation I from its definition.
Update the total cost by the cost of creating I.
Update the available storage size by subtracting the size of relation I.
Set relation I explicit.

xii) If the last reference has been processed go to (xiii) else go to (iii).

xiii) Print the dynamic storage size, the total cost, the number of successes, the number of page swaps and the number of page faults.

xiv) STOP.

5.4 The Results

Figure 3.3 shows the performance of all algorithms against a uniform reference string of length 500. This curve and the following ones illustrate the variation in processing cost as the amount of dynamic storage space is increased. Each cost curve is monotonically decreasing since the frequency of page faults lessens as more relations are maintained explicitly. When the dynamic store is large enough to accommodate all the pages, all algorithms perform as well as the ideal. Figure 3.4 shows the same results normalized against the minimum cost curve. The cost axis

now represents the cost relative to the ideal.

Figure 3.4 illustrates clearly the need to consider size and cost in making page replacements. The standard algorithms for uniform pages, LRU and LFU, perform essentially at the level of random replacement. They continuously diverge from the ideal until the storage size is large enough to accommodate all the pages, when their curves become discontinuous and drop suddenly to the ideal performance. The two parameter-sensitive methods, LECS and LEC, on the other hand, actually begin to approach ideal performance after an initial divergence.

Figure 3.5 shows the cost curves for a string of low cost weighting, type (ii), and figure 3.6 shows the cost of the algorithms relative to the minimum achievable cost. The variation in the cost of replacement between the algorithms is not large because the penalty of a wrong choice has a low cost.

In figures 3.7 and 3.8 at large dynamic storage sizes the LEC behaved better than the LECS. This is because:

(1) the size parameter became less significant.

(2) the reference string is weighted for high cost,

and (3) the LEC concerns itself with the cost only.

In figures 3.11 through to 3.15 the algorithms are compared against an idealised algorithm "The Preferred Set" which will be explained later. With a longer string in these cases the parameter-sensitive algorithms give a clearcut reduction

in cost. The performance of ARB, LRU and LFU is nearly the same except in figure 3.13 where the high cost penalties lead to distinct differences in the performance of the algorithms.

Essentially the same conclusions are reached. The inclusion of size and cost information in the replacement strategy results in a clearcut advantage in performance. In fact, a particularly unlucky set of relation characteristics can lead to worse-than-random performance by LFU or LRU. This occurs in case (v), figures 3.9 and 3.10, because relations requiring large storage area but having low cost tend to be maintained in the store, whereas the storage space could better be used to hold high-cost, smaller relations, in spite of the lower occurrence rate of the latter.

5.5 The Success Function

In figures 3.14 and 3.15 the success function [Mattson et al 1970] is plotted against the dynamic storage size for all the above algorithms with input reference string of types (vi) and (viii) respectively. The success function is the proportion of times a requested page is found in the storage (explicit) over the total length of the string, i.e. hit ratio.

In figure 3.14, if an algorithm (e.g. LFU) maintains the frequently requested, less costly relations, this will lead to a larger number of successes but the cost of computing the less frequent, more expensive relations may be high enough to offset the gain from the successes. Thus in contrast to conventional paging, a high number of successes is not necessarily a merit for the algorithm.

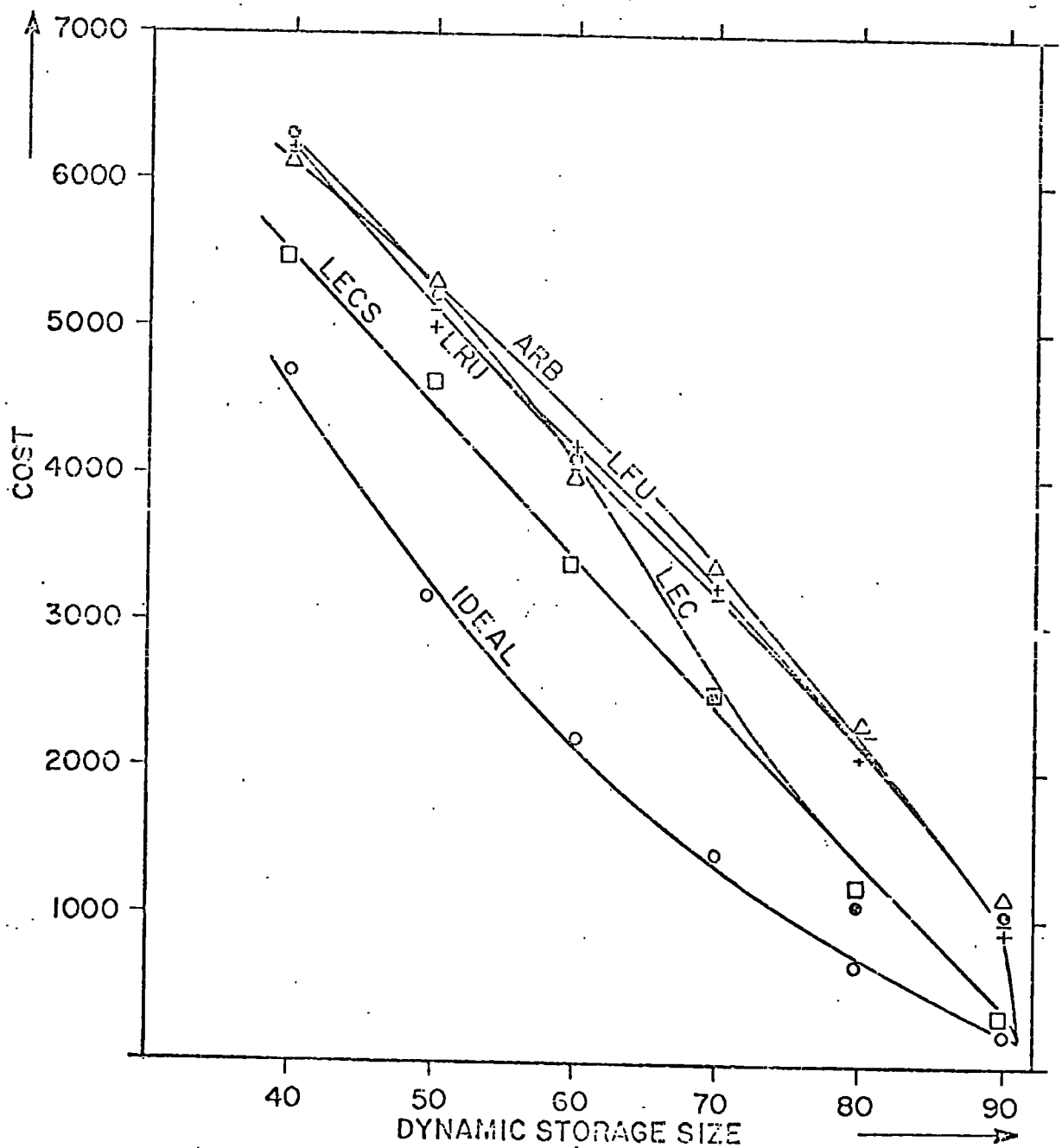


FIGURE 3.3 : Cost/Buffer size for uniform distribution (string type (i))

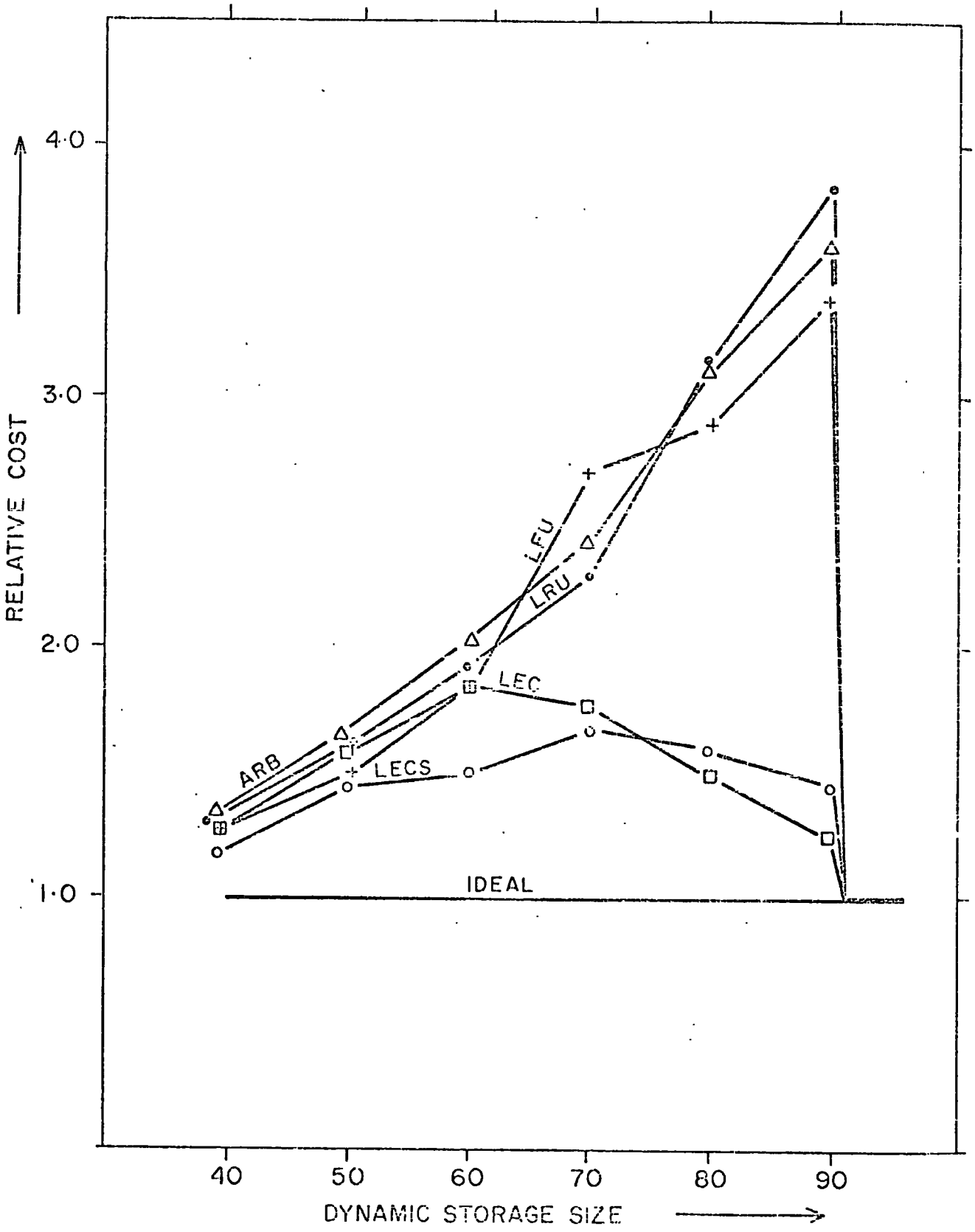


Fig 3·4 Relative Cost/Storage Size for uniform distribution (string type (i))

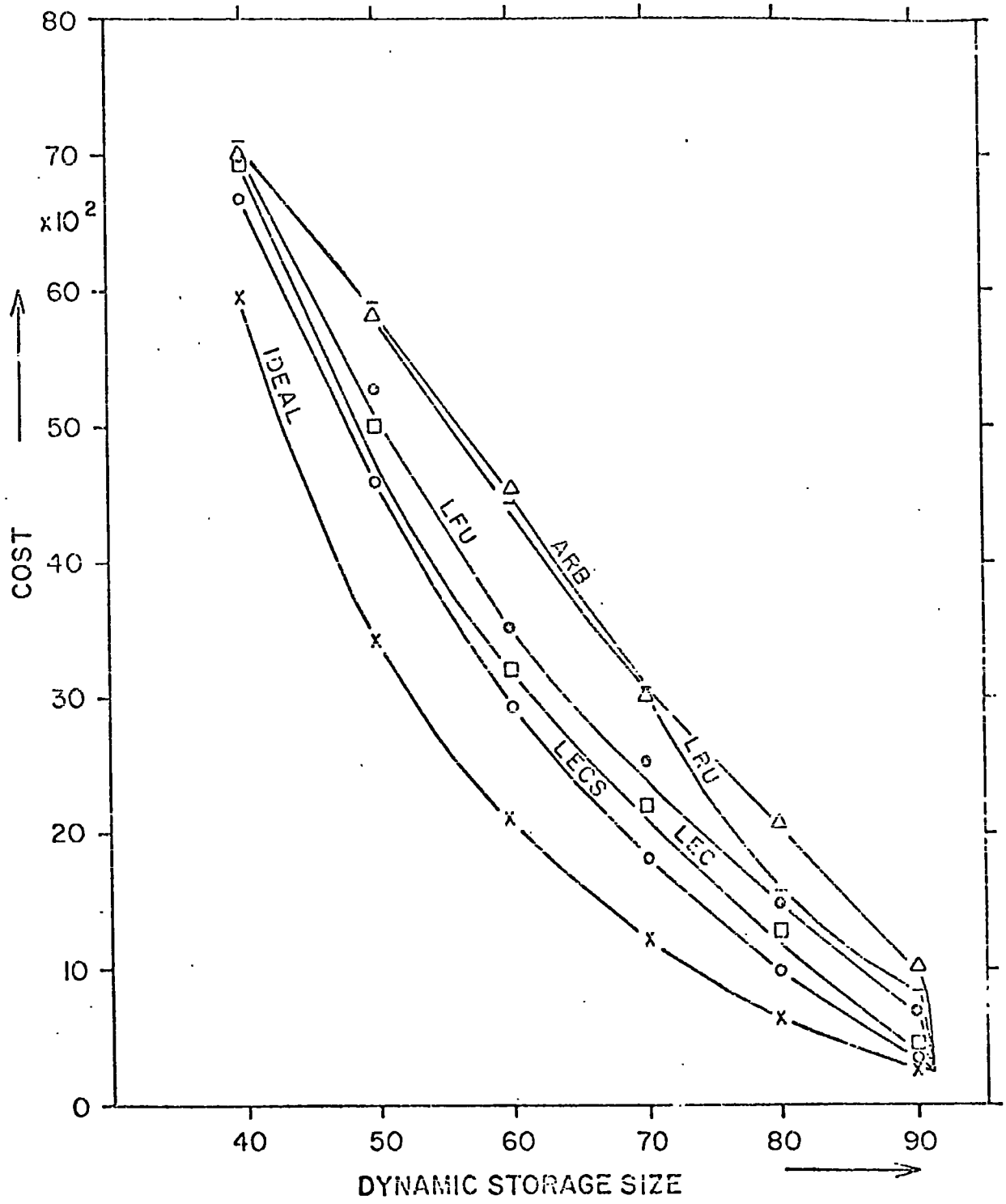


FIGURE 3.5 Cost curves for low cost weighting string length = 500

FIGURE 3.6 Relative cost for Figure 3.5

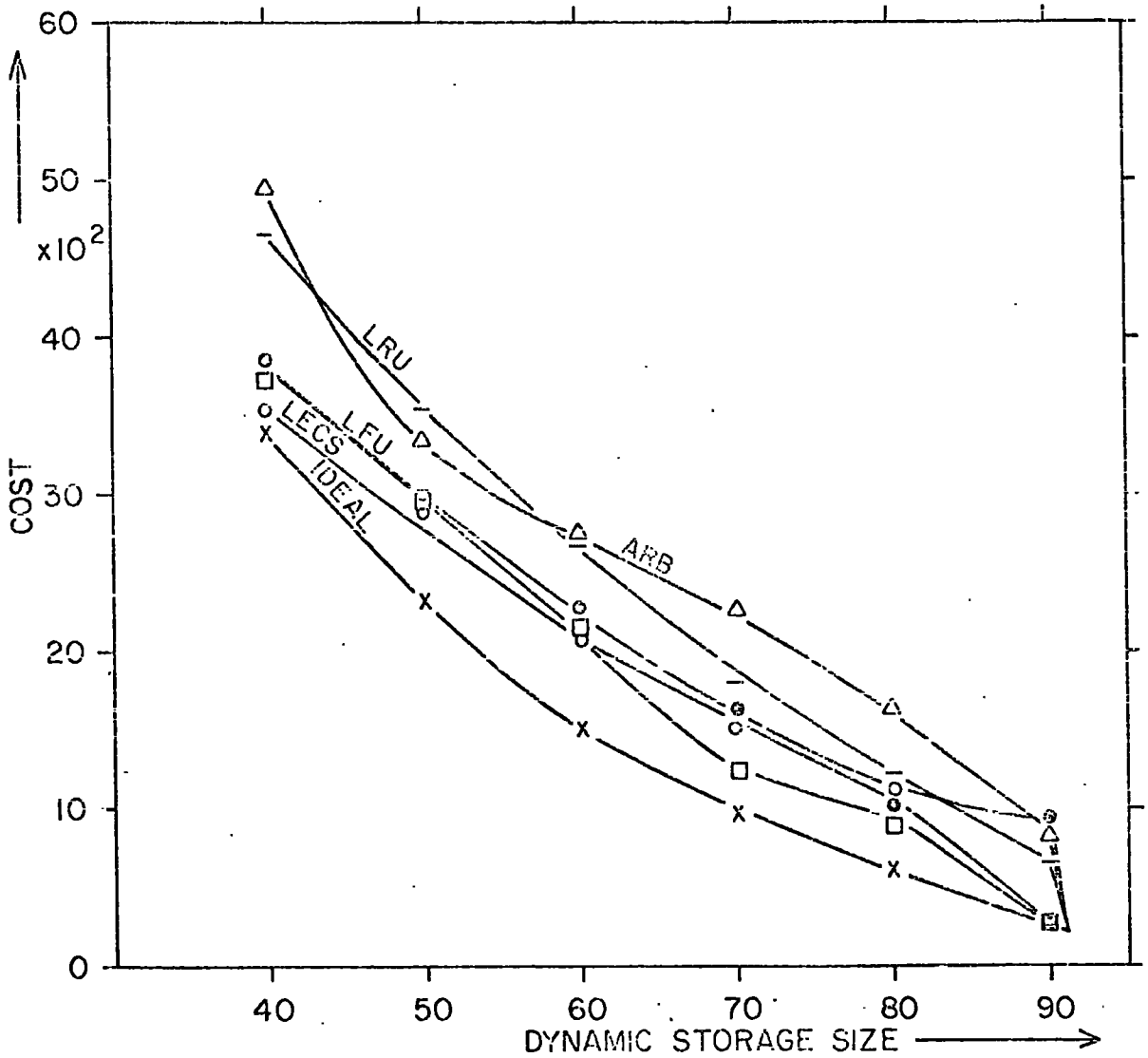


Fig 3·7 High cost weighting (string type (iv))

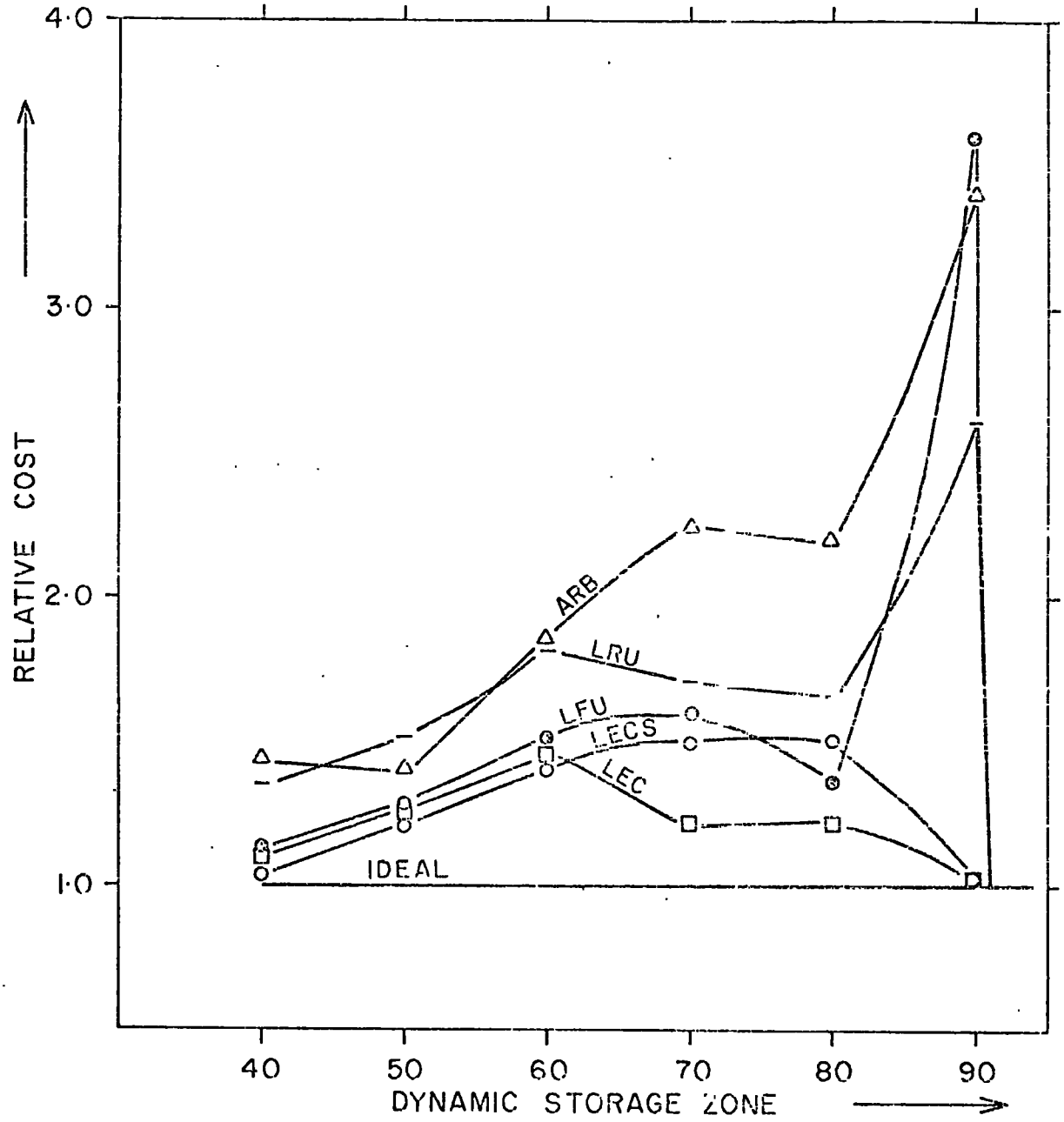


Fig 3·8 Relative cost for fig 3·7

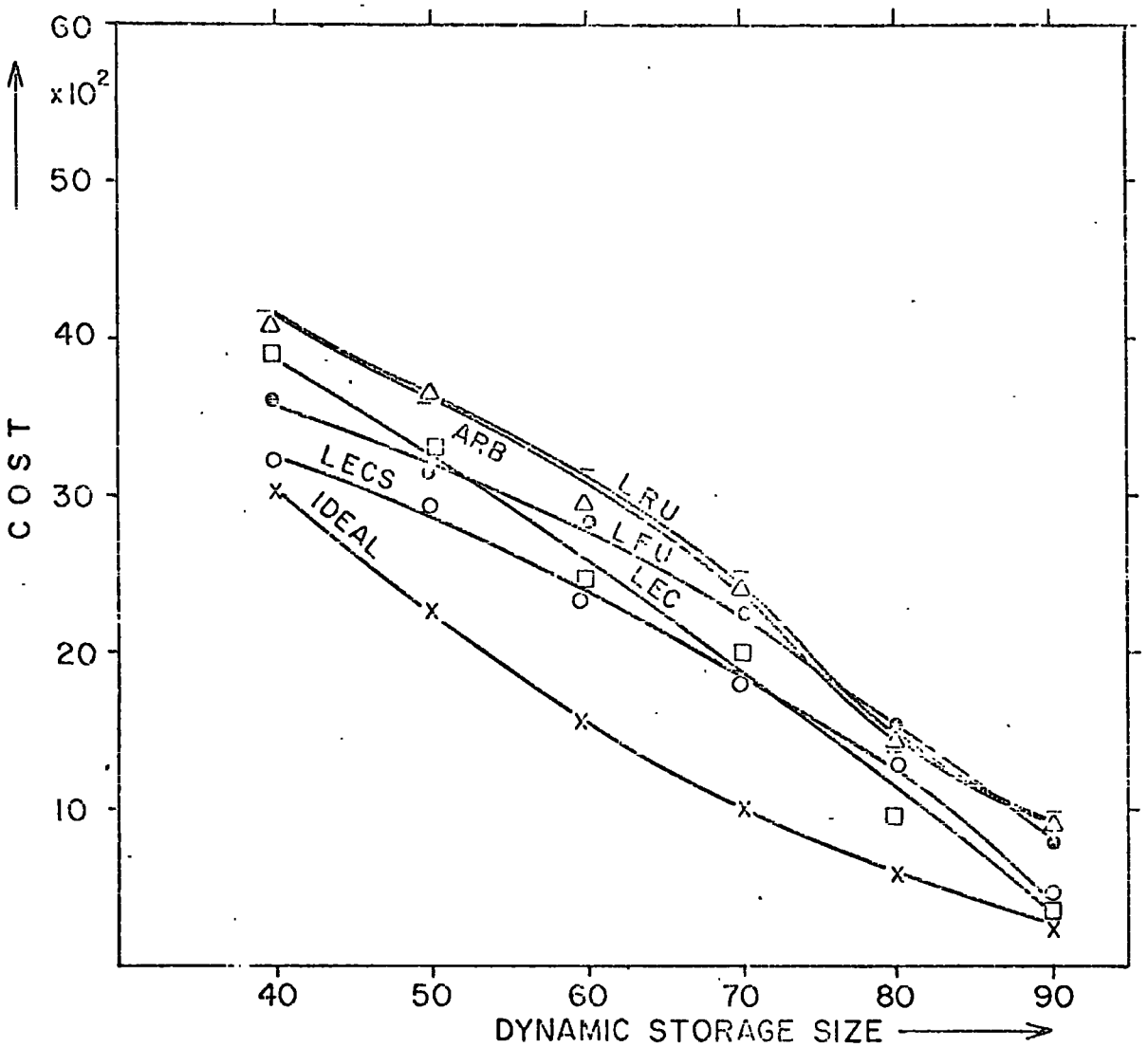


Fig3·9 COST/BUFFER SIZE FOR SMALL SIZE WEIGHTING (string type(v))

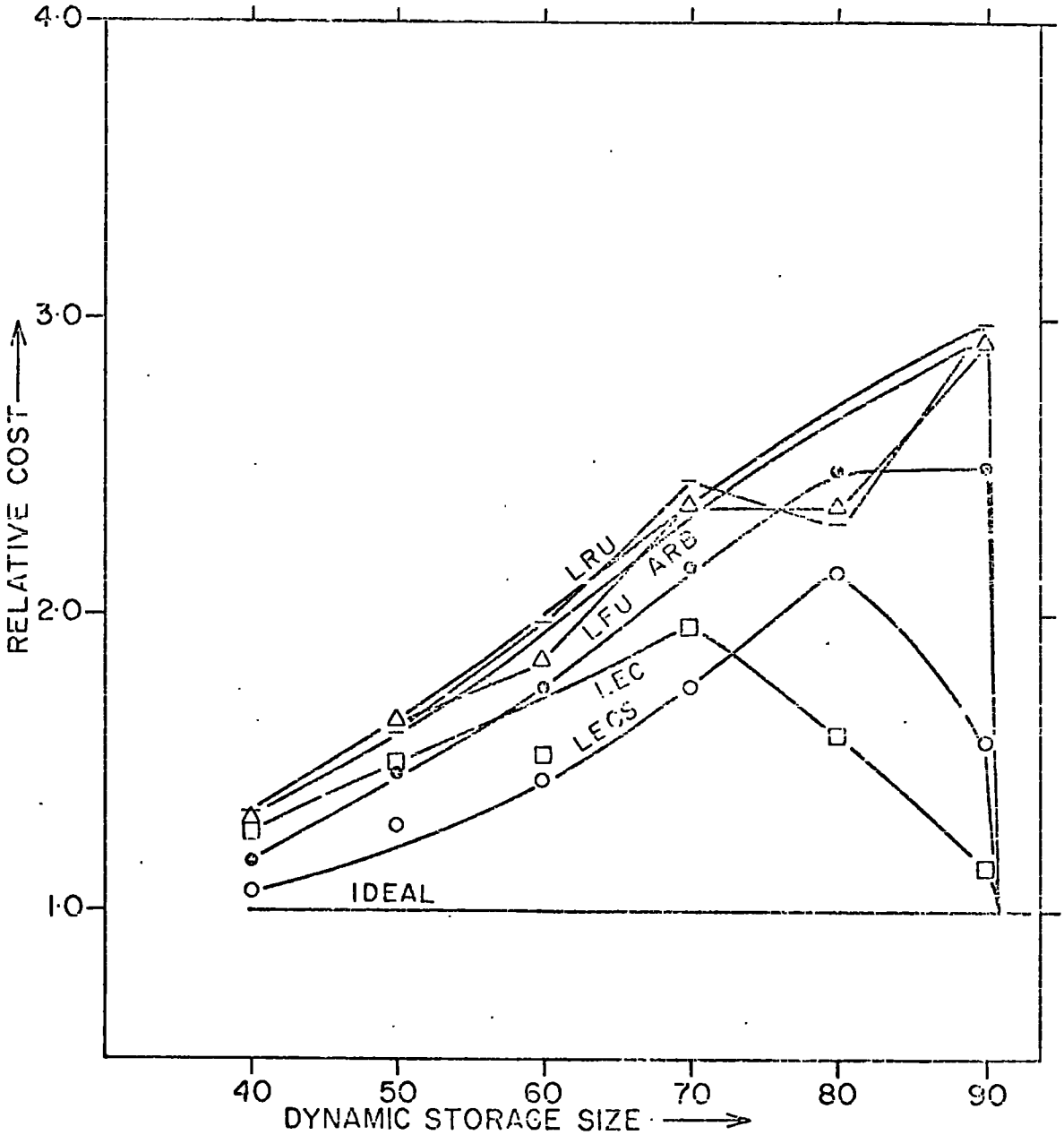


Fig 3·10 Relative cost for fig 3·9

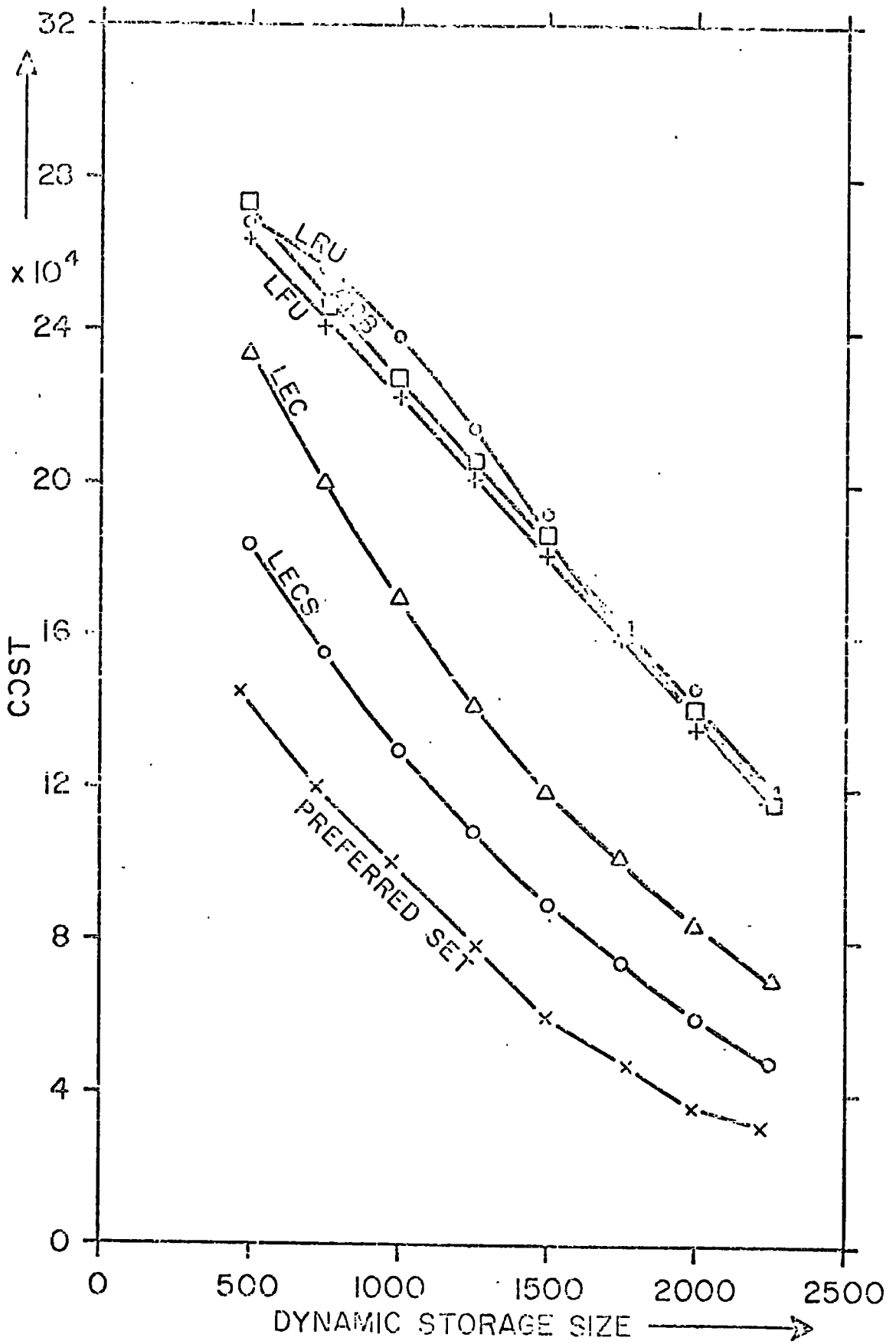


Fig 3·11 Cost/Buffer size for low cost weighting (siring type (vi))

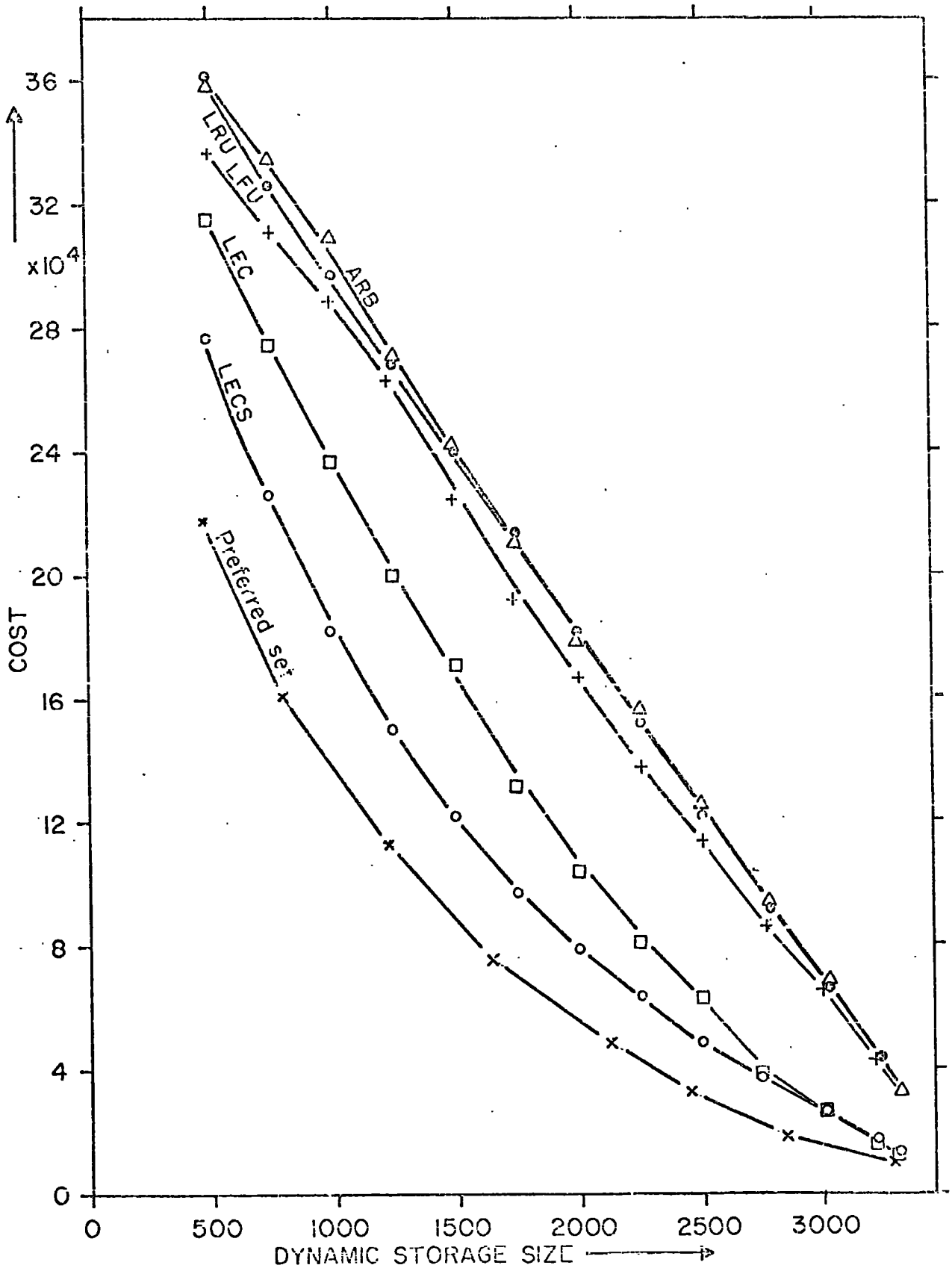


Fig 3·12 Cost/Buffer size for large size weighting (string type (vii))

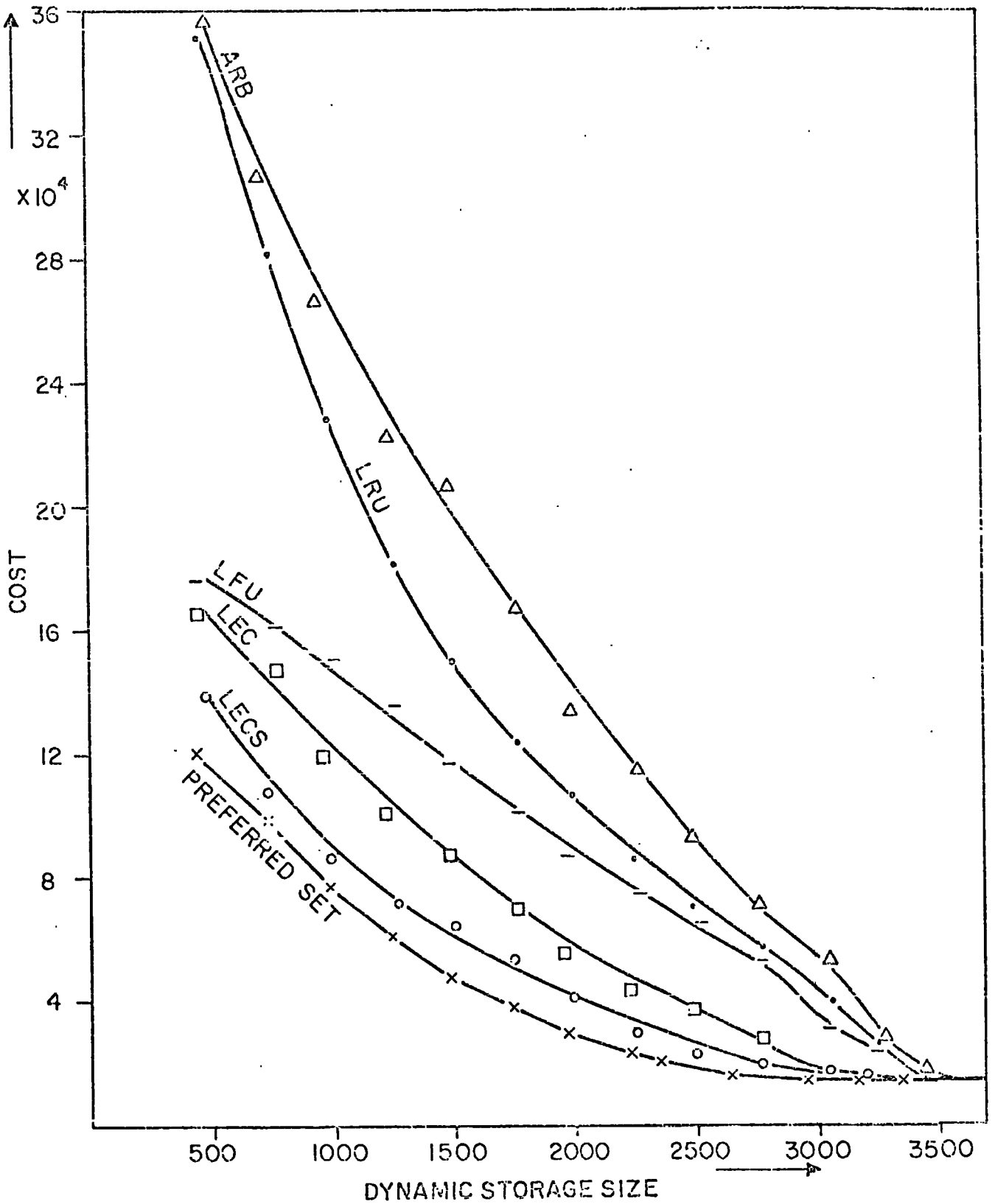


Fig.3.13 Cost/Buffer Size for high cost weighting (string type (viii)

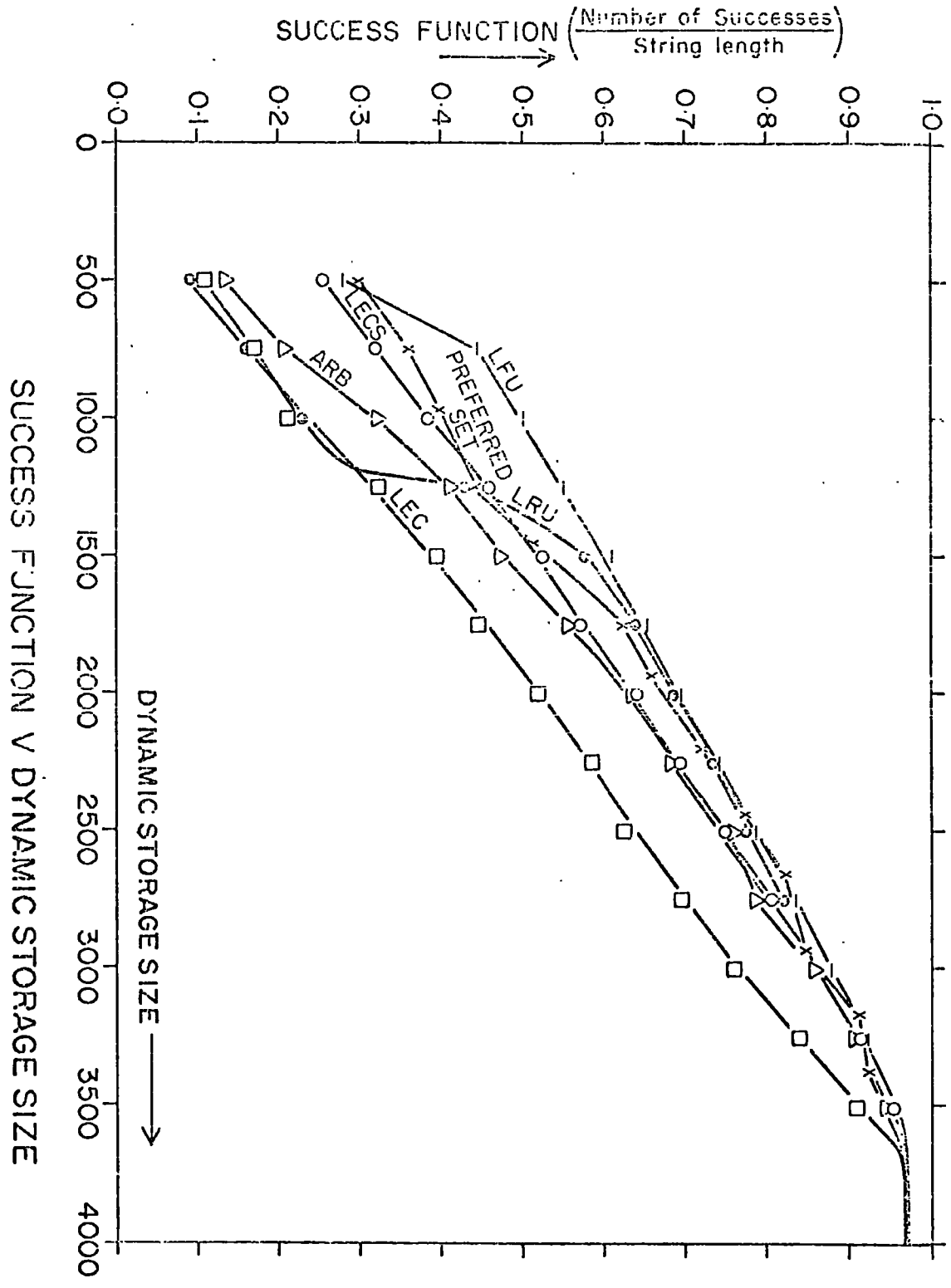


FIGURE 3.14 Success function for string type (vi)

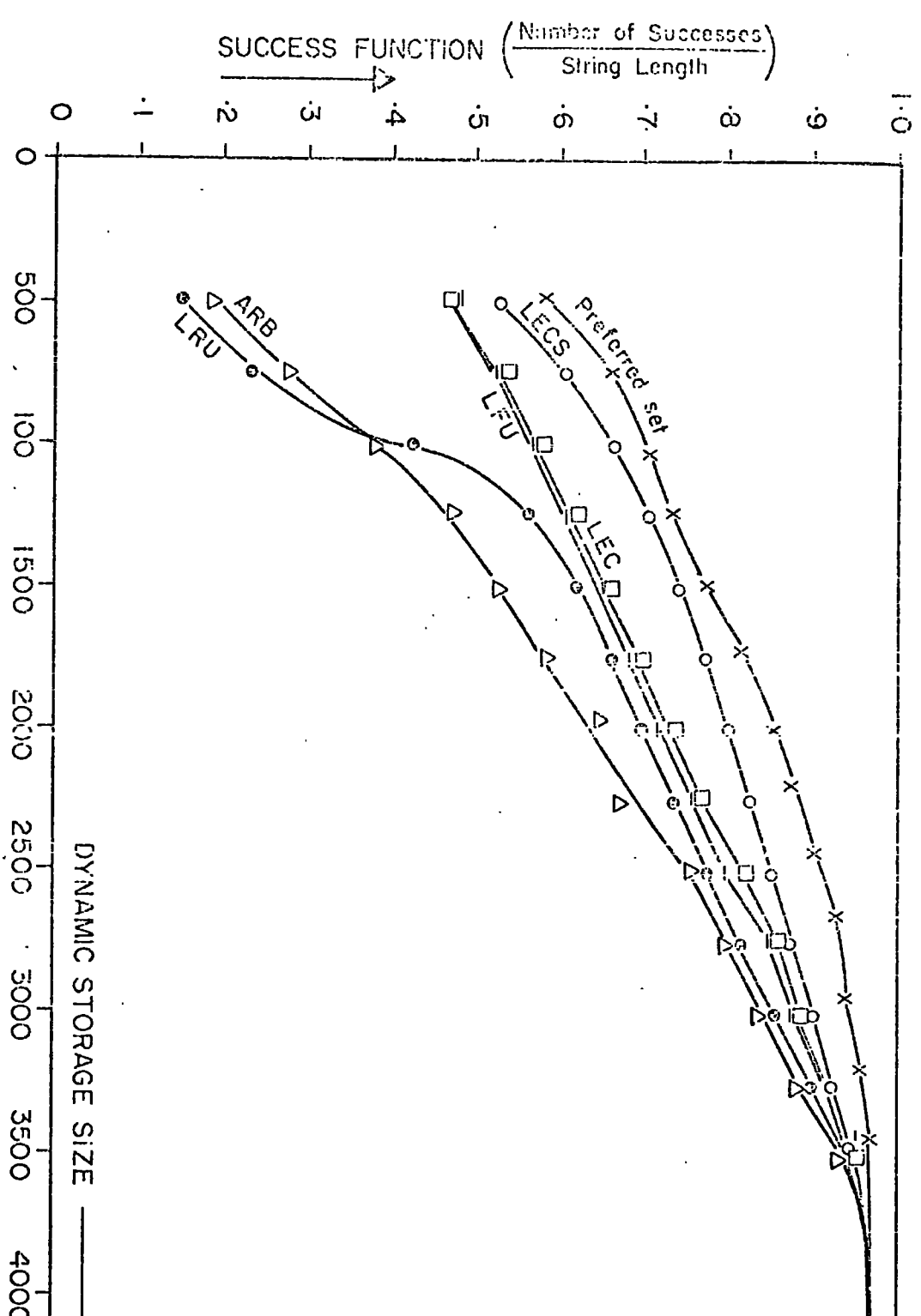


Fig. 3·15 SUCCESS FUNCTION V. DYNAMIC STORAGE SIZE (string viii)

However, in figure 3.15 the high cost relations are referenced more frequently. The LECS maintains the expensive relations and hence it gives a high hit ratio.

6. The Preferred Set

The experimental results obtained above consistently favour the LECS algorithm over the other replacement techniques. However, they leave open the question: is there a better method, or does the LECS rule indeed yield least cost among demand replacement strategies?

This question cannot be answered in general, but there is an answer to a related theoretical question, and the result furnishes additional understanding of LECS and its limitations.

The question is formulated as follows:

Given a fixed amount of storage and a reference string, which relations should be permanently stored such that the cost of fulfilling the requests over the given time interval is minimum?

This question has been answered by Casey [Casey and Osman, to be published] and the resulting set of relations is known as "The Preferred Set". The solution is given in Appendix 5.

This algorithm does not guarantee an optimal set for all storage sizes but it serves as a useful bound on the cost reduction achievable.

The preferred set criterion is:

i) For each relation k form the ratio $Z_k = \frac{U_k \cdot C_k}{S_k}$

(where U_k, C_k and S_k are the usage count, the cost and the size of relation k respectively).

ii) Arrange the Z_k in decreasing order and keep in storage those that have the highest Z_k.

This is precisely the LECS technique. The LECS does not know the string of requests beforehand. It also employs estimates for the frequency rather than true probabilities.

Thus the LECS method is an approximation of the optimal technique for managing the dynamic storage in response to randomly (i.e. independent successive occurrences) occurring references.

For comparison purposes, the cost performance of the preferred set is plotted along with the LECS and other algorithms in figures 3.11 through to 3.15.

7. Summary

It has been shown that the task of automatic workspace management in a relational data base environment is a generalisation of the conventional problem of assigning space in a given buffer area among a set of data pages of fixed size. The management objective is to minimise the total cost of page faults, where the cost of a fault depends on the page identity.

Experiments in this extended environment indicate that replacement algorithms previously considered for uniform-pages will not perform adequately. In some tests the

well-known LRU and LFU algorithms give results about equivalent to random replacement. An appreciable reduction in cost can be achieved by weighting the replacement test criterion with the page cost and size parameters in an appropriate manner.

A further justification of the intuitively derived LECS replacement criterion has been demonstrated in the "preferred set" analysis. The LECS criterion ranks the pages for replacement in the same order as a system which is stocking a buffer so as to minimize the expected cost of meeting the next data request.

A relational data base is the prime example of a system which realises these generalised assumptions of a paging model. However, it is worthy of note that some operating systems, for example the Burroughs B5500 MCP [Burroughs 1966] have employed non-uniform page sizes and so the results obtained here may possess wider application. In general, the problem of allocating resources under storage type constraints occurs in many forms in a number of systems context. Thus the results obtained may be capable of wider application.

Chapter 4

FURTHER GENERALISATION OF THE REPLACEMENT ALGORITHMS

Introduction

This chapter is an extension to the previous chapter. The following topics are discussed:

(1) Chained dependency:

The LEC and LECS algorithms are extended to account for the case when relations are defined on other defined relations.

(2) Rapidly changing usage patterns:

If the usage pattern is such that only a small subset of relations is used over a period of time and then suddenly another subset becomes active and so on, then under such circumstances the frequency of reference for each relation does not give sufficient indication of the probability of that relation being referenced in the future. The attempts to solve this problem are explained.

(3) The principle of defined relations is justified on cost basis.

(4) A method for estimating the probability of a relation remaining implicit.

1. The Dependence among Relations

Up to this point we have assumed that the cost of accessing a defined relation takes on one of two values: a negligible quantity if the desired relation is available in the dynamic

storage area, or else some positive quantity if it must be reconstructed from base data.

More generally, however, a relation may be defined on other defined relations. The dependencies among relations may be represented as a directed graph (Figure 4.1). Here recursive (circular) definitions are disallowed, i.e. there must be no cycles in the graph of dependencies. Base relations need not be shown explicitly in the graph, since they can be made available at negligible cost.

Consider now a relation X having the dependencies shown in Figure 4.1. The cost of forming X in response to a request is contingent on whether its subcomponents A and B are in explicit form. Let C_X be the cost of calculating X from A and B, and let C_A, C_B be the cost for creating A and B from base data. We may write the cost function for X as:

$C(X) = 0$ if X is explicit

$= C_X$ if A and B are explicit but X is not

$= C_X + C_A$ if B is explicit but A, X are not

$= C_X + C_B$ if A is explicit but B, X are not

$= C_X + C_A + C_B = T(X)$ if none of A, B, X are explicit

1.1 Generalisations of LEC and LECS

Now let us extend the LEC and LECS algorithms to the case of such chains of dependence. The LEC criterion depends on the

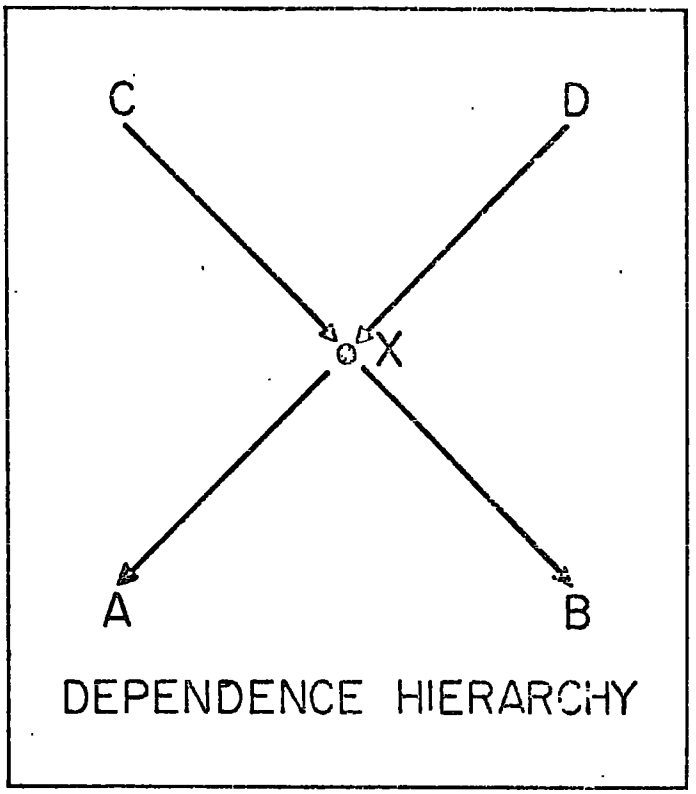


Figure 4.1

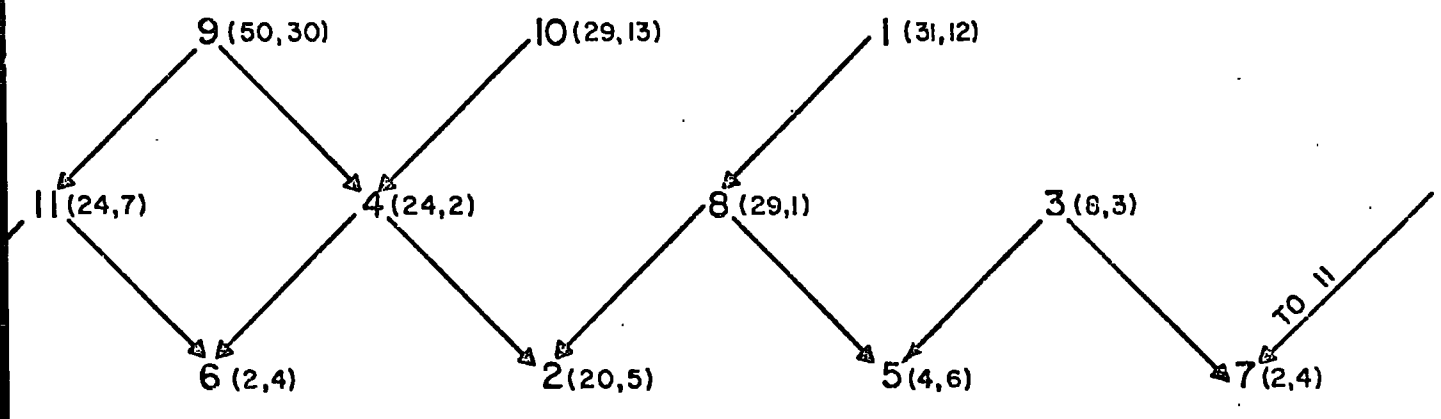


Figure 4.2

product of the number of references (usage count) for a relation times its cost. We will, therefore, extend the concepts of the usage count and the cost to account for these dependencies.

In Figure 4.1 relation X will be requested to help in computing relation D whenever D is requested and found not explicit. The usage count for relation X must be extended to cover requests for X for the sole purpose of computing dependent relations.

Thus, the expanded usage count for relation X from time = 0 to time = t, denoted F_X^t, is defined as the number of times X is directly requested, U_X^t, plus the number of times X is requested in order to assemble one of its dependent relations (C or D).

Let us also define the occupancy of relation D (θ_d^t) as the ratio of times D is in the dynamic area to the number of requests for all the defined relations. It will be assumed that the probability of a request for a relation is independent of the presence or absence of that relation in the dynamic store.

The proportion of times D is not in the dynamic area

$$= (1 - \theta_D^t)$$

The number of times D is requested and found implicit

$$= U_D^t \cdot (1 - \theta_D^t)$$

Hence, F_x^t may be expressed as:

$$F_x^t = U_x^t + U_D^t \cdot (1 - \theta_D^t) + U_C^t \cdot (1 - \theta_C^t)$$

This expression may be used as an LFU criterion which accounts for dependencies (Improved LFU).

The cost of computing X when it is implicit depends on whether A and B are explicit or implicit. An expression for the cost can be found by multiplying each of the four possible values listed above by its probability.

Hence,

$$C(X) = [T(X) - C(A) - C(B)] \cdot Pr(A,B) + [T(X) - C(B)] \cdot Pr(\bar{A},B) + [T(X) - C(A)] \cdot Pr(A,\bar{B}) + T(X) \cdot Pr(\bar{A},\bar{B})$$

where:

$$Pr(A,\bar{B}) = \theta_A^t \cdot (1 - \theta_B^t) \quad , \quad Pr(\bar{A},B) = (1 - \theta_A^t) \cdot \theta_B^t$$

$$Pr(A,B) = \theta_A^t \cdot \theta_B^t$$

$$Pr(\bar{A},\bar{B}) = (1 - \theta_A^t) \cdot (1 - \theta_B^t)$$

Similarly,

$$C(D) = T(D) \cdot Pr(\bar{X}) + [T(D) - C(X)] \cdot Pr(X)$$

Therefore, for LEC we use the product $F_i^t \cdot C(i), (i=1,2,\ldots,n)$, which reduces to $U_i^t \cdot C_i$ if there are no dependencies. Similarly for the LECS we have the expression $\frac{F_i^t \cdot C(i)}{S_i}$ which again reduces to $\frac{U_i^t \cdot C_i}{S_i}$ for the case of no dependencies.

1.2 Experiments

1.2.1 The Simulation Experiments

Dependencies have been assumed to exist among the previously mentioned 11 relations in the manner shown in Figure 4.2. Simulation experiments have been conducted as described in Chapter 3. In order to evaluate the cost of creating a relation R the list of relations which define R is examined. A recursive procedure finds which of the defining relations is explicit and accordingly it evaluates C(R).

The reduction in cost due to the presence of the relations defining relation I in their explicit form is evaluated as follows:

```
integer procedure r(i); comment recursive;
integer i,j,k,it,h;
comment: i      is the relation for which the cost reduction
                  due to explicit defining relations is to
                  be calculated.
         T(k)   is the creation cost of relation k.  It is
                  the reduction due to explicit defining
                  relations.
         dep(i) is a global array of the number of relations
                  on which i is dependent.
         d(i,h) is the h^th relation on which i is dependent;
if dep(i) = 0 then r:= 0;
       it:= 0;
for    h := 1 step 1 until dep(i) do;
begin
    k:= d(i,h);
```

```
    if status(k) = 'explicit' then it:= it + T(k);
    else it:= it + r(k);
end;
    r:= it;
end;
```

At any instant of time when a relation is references, the occupancy for each relation that happen to be explicit is updated by one.

When the LEC or LECS criterion is applied, a recursive procedure evaluates the expected cost of each relation using the expression for the expected cost.

Similarly, the Ideal replacement algorithm was adjusted to account for dependencies.

1.2.1 The results

The string of type (i) (Chapter 3) was used to investigate the effect of dependency on the behaviour of algorithms. This is illustrated in Figure 4.3.

Comparing with Figure 3.3 the cost has dropped for all algorithms due to the reduction caused by the dependencies. The LRU and LFU algorithms do not take into account the effect of the dependencies, thus their cost of replacement is relatively high. The LECS is near enough to the ideal. In fact the size parameter is very important in the case of dependencies because the cost parameter is generally reduced. E.g. when the dynamic store sizes are 80 and 90 units the ideal replacement strategy, found by inspection, is to maintain all the

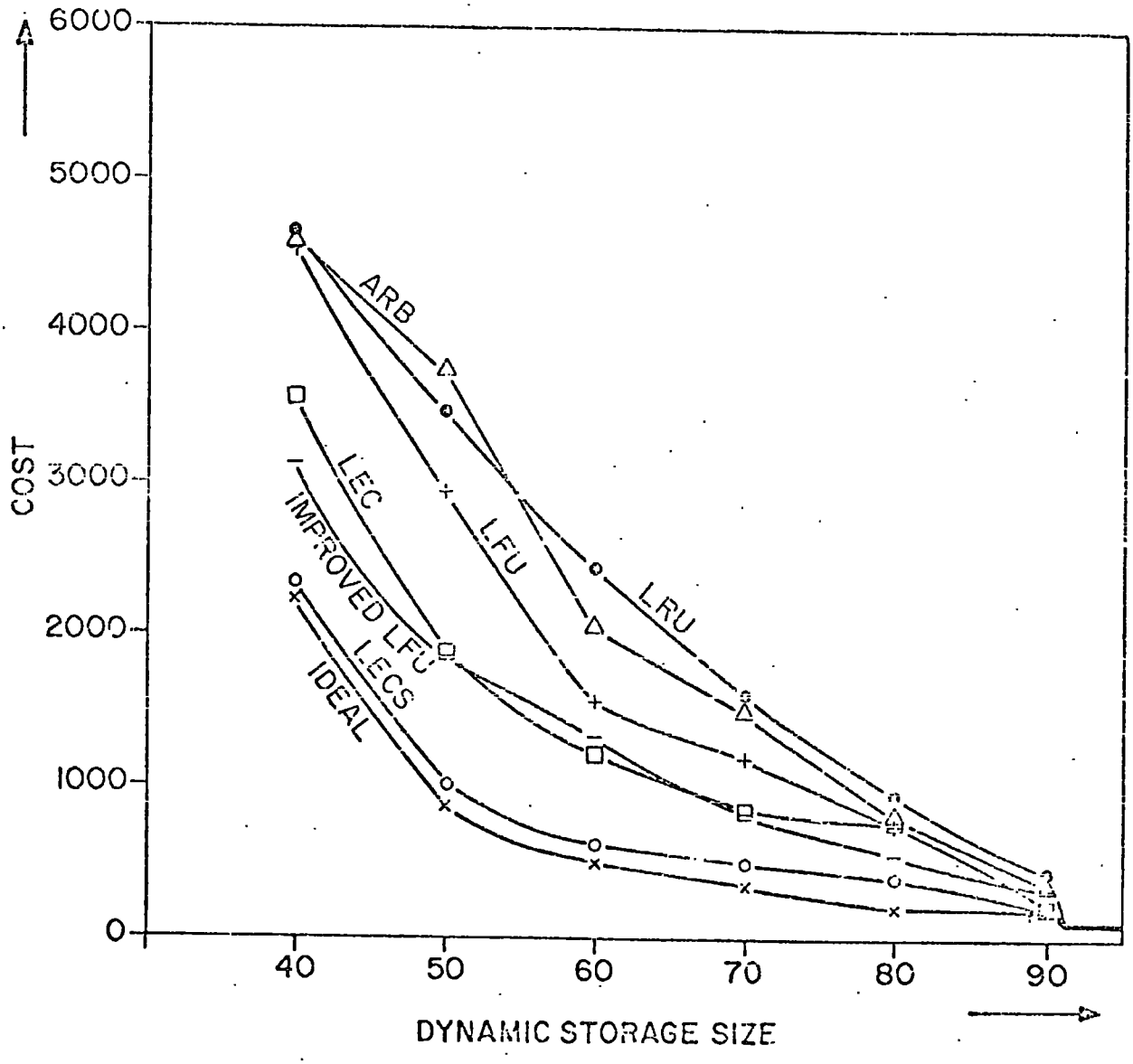


FIGURE 4.3 Cost/storage size for uniform distribution with dependencies

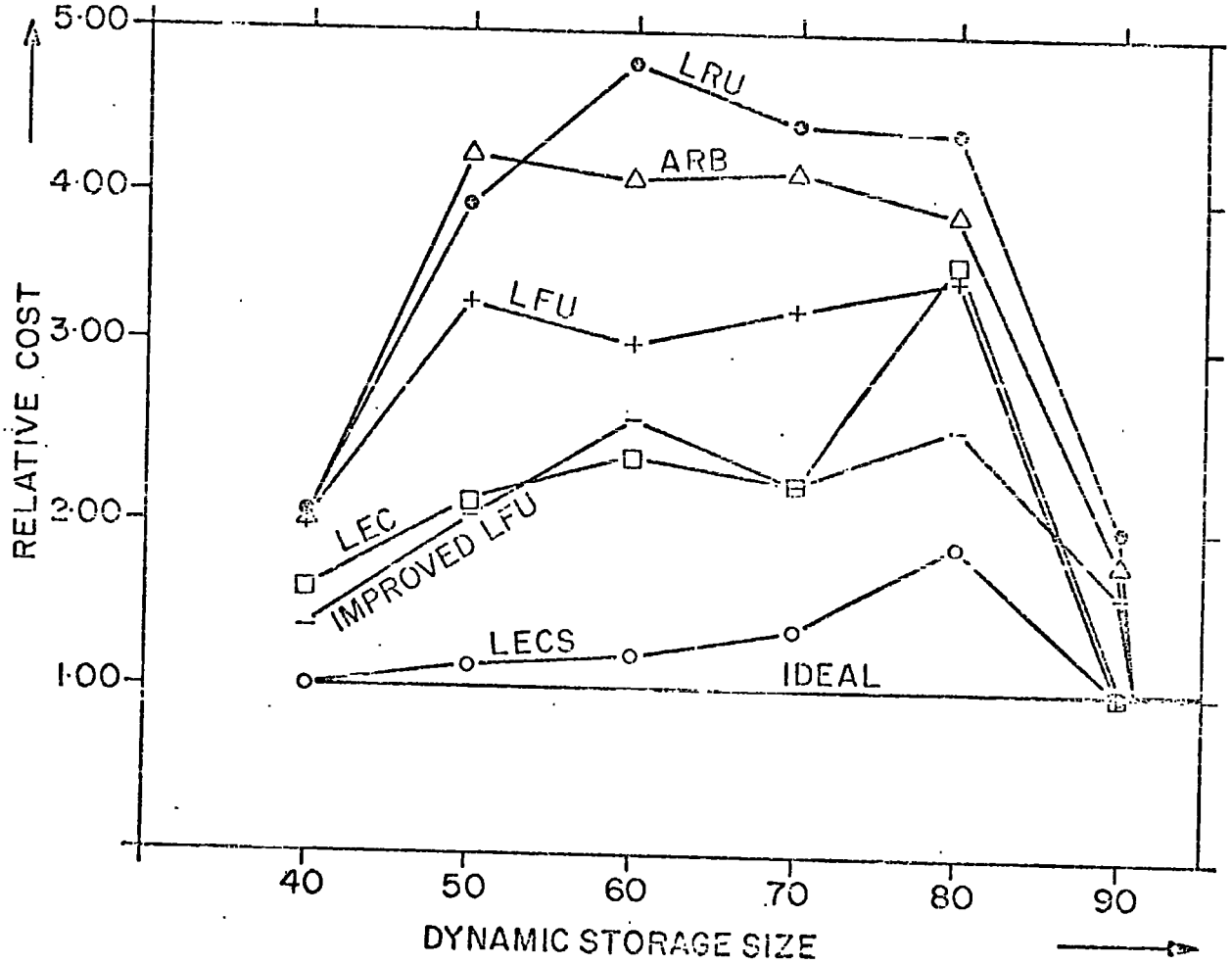


FIGURE 4.4 Relative cost for Figure 4.3

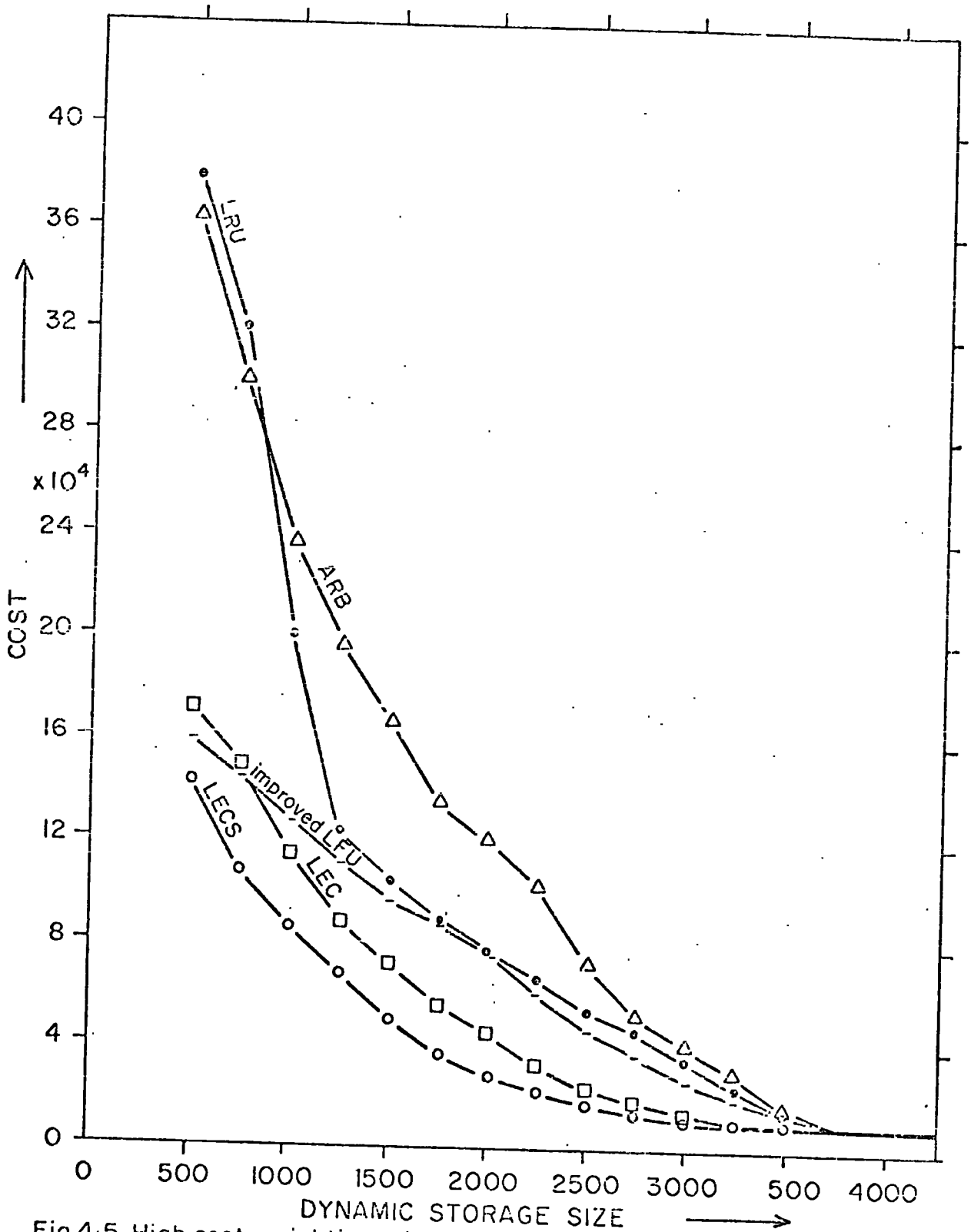


Fig 4·5 High cost weighting with dependancy

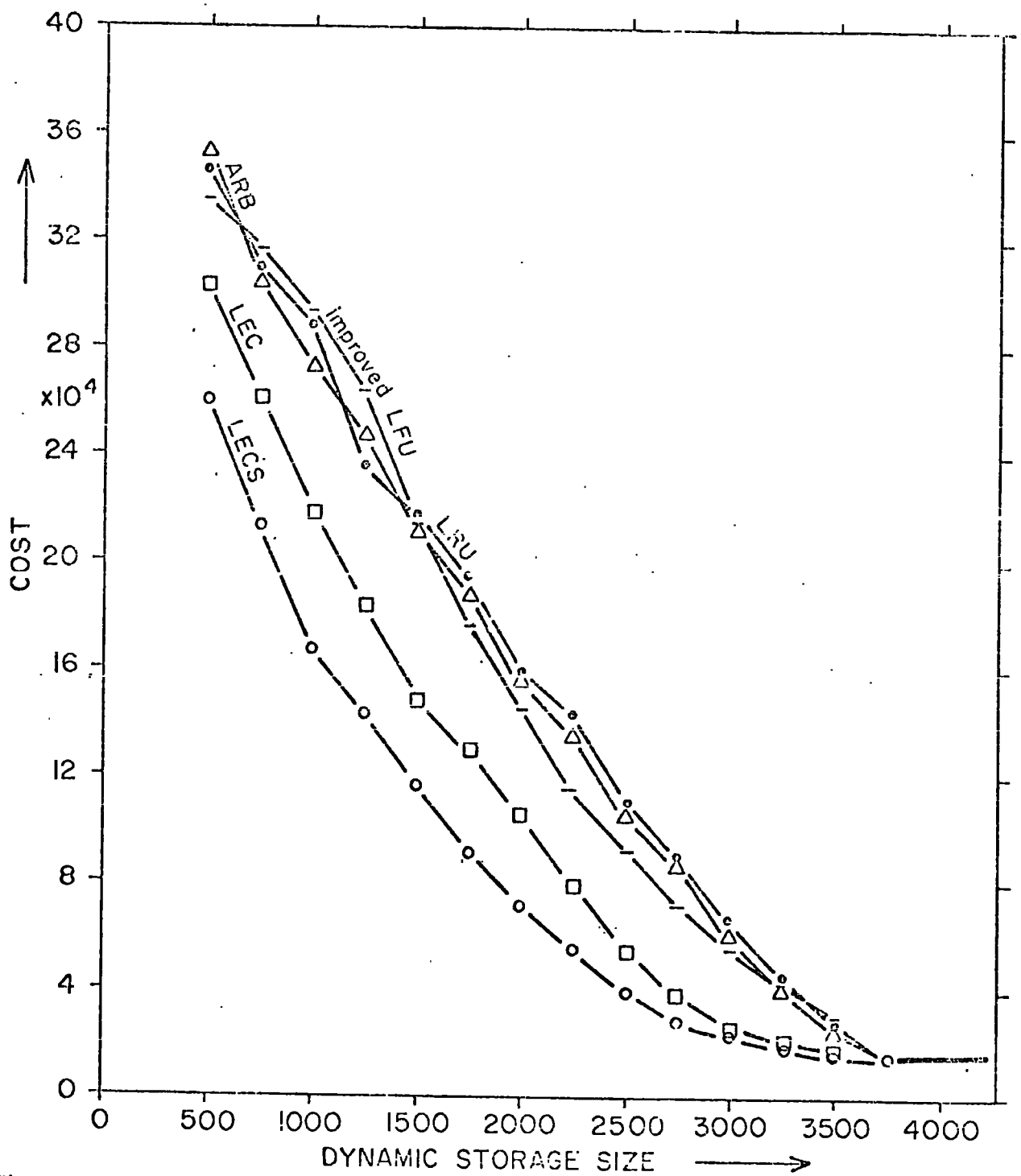


Fig 4·6 Large size weighting with dependency

relations explicit except for relations 9 and 1 which are to be swapped in the dynamic storage. This is because their cost, when other relations are in the dynamic storage, is small and their size is large and no other relation depends on them. It is interesting to note that this is the exact strategy followed by LECS after the initial fluctuations. This is because it takes the size into consideration along with the other parameters.

The relative cost for Figure 4.3 is plotted against dynamic storage size in Figure 4.4. Comparing with Figure 3.4 all the algorithms converge towards the ideal behaviour at larger dynamic storage sizes. There is no abrupt fall towards the ideal as in Figure 3.4. This may be due to the fact that some of the larger relations that formerly used up storage space have now relatively smaller cost and so the penalty for keeping them implicit is reduced.

Dependancies were also assumed to exist among 50 of the 100 relations of Chapter 3. Seven groups of dependent relations were formed. The reference strings of types (vii) and (viii) were run against the 100 relations. The cost curves are shown in Figure 4.5 and 4.6 respectively.

In Figure 4.3 through to 4.6 the significant improvements of the "Improved LFU" over the "LFU" reflects the validity of the model adopted for estimating the frequency of dependent relations.

The curves also show that the LEC does not give significant

improvement over the "Improved LFU" because with dependencies the influence of the cost parameter is reduced.

1.3 Conclusion

The LECS method can be extended to the case of data that is defined on base data through a sequence of operation steps, and where the results of these steps may themselves be available in the system at a given time.

The experiments have shown that the extended LECS algorithm is still the best of those described so far which accounts both for the relational dependencies and for the time-varying contents of the dynamic store.

2. Rapidly changing reference patterns

The present LECS algorithm weighs references to a relation equally regardless of the point in time at which the references took place. This may not give a satisfactory prediction of the coming requests in a heavily used data base with a rapidly changing reference pattern.

Here, it is sought to extend the LECS to account for strings of requests having a rapidly changing pattern. For this purpose a string of requests was generated and it was run against the three simulation models. The results are reported below:

2.1 Generation of reference strings

In these strings a subset of the defined relations is referenced for a period of time, then abruptly another subset is referenced and so on.

Strings of 600 requests for 20 relations were generated. Each string represented one of the combinations of the period length and the number of relations in a subset (working set size).

Given a period of P references and a working set size of w relations, the references are generated as follows:

i) pick w relations at random from the 20 relations.

ii) generate a random reference to one of the w relations. Repeat this P times.

iii) if the number of references generated is less than the required length, go to (i).

2.2 The Models

These models adopt different ways of weighing the frequency term of the LECS criterion.

The cost of running a string of requests against a model was estimated by finding the area under the cost curve, i.e. the cost is proportional to the product of the processing time and the size of the storage space. Using the familiar trapezoidal rule, the total cost of running the string is calculated from the costs of running the string at 12 storage sizes.

i.e. the cost due to an algorithm

$$= h \left(\frac{C_1 + C_{12}}{2} + C_2 + C_3 + \ldots + C_{11}\right)$$

$$\propto \frac{C_1 + C_{12}}{2} + C_2 + C_3 + C_4 + \ldots + C_{11}$$

where C_i is the cost of running the string of requests against the model for the i^{th} storage size.

h is the fixed interval between two storage sizes.

2.2.1

One possible model for weighting references at different times is as follows:

For relation i update f_p^i periodically at the end of the p^{th} period by the weighted number of references occurring during that period.

$$f_p^i = \alpha \cdot f_{p-1}^i + (1 - \alpha)\, u_{p-1,p}^i \qquad f_o^i = o \qquad 0 < \alpha < 1$$

where

α = the attenuation coefficient for weighting past references

$u_{p-1,p}^i$ = the number of references within the p^{th} period

f_{p-1} = total number of weighted references until the p-1 period

The parameters α and the length of the period have to be estimated. The choice of α and a period length to suit different patterns is difficult. Moreover, there is usually a phase shift between the period of pattern change of the string and the period in the model.

Each generated string of requests was run against the above model using a range of periods. For each period α was varied between 0.01 and 0.99. The values of α and the period that give the minimum cost are kept.

An attempt was made to find some relation between α_{opt} and the parameters of the string, so that given a certain string

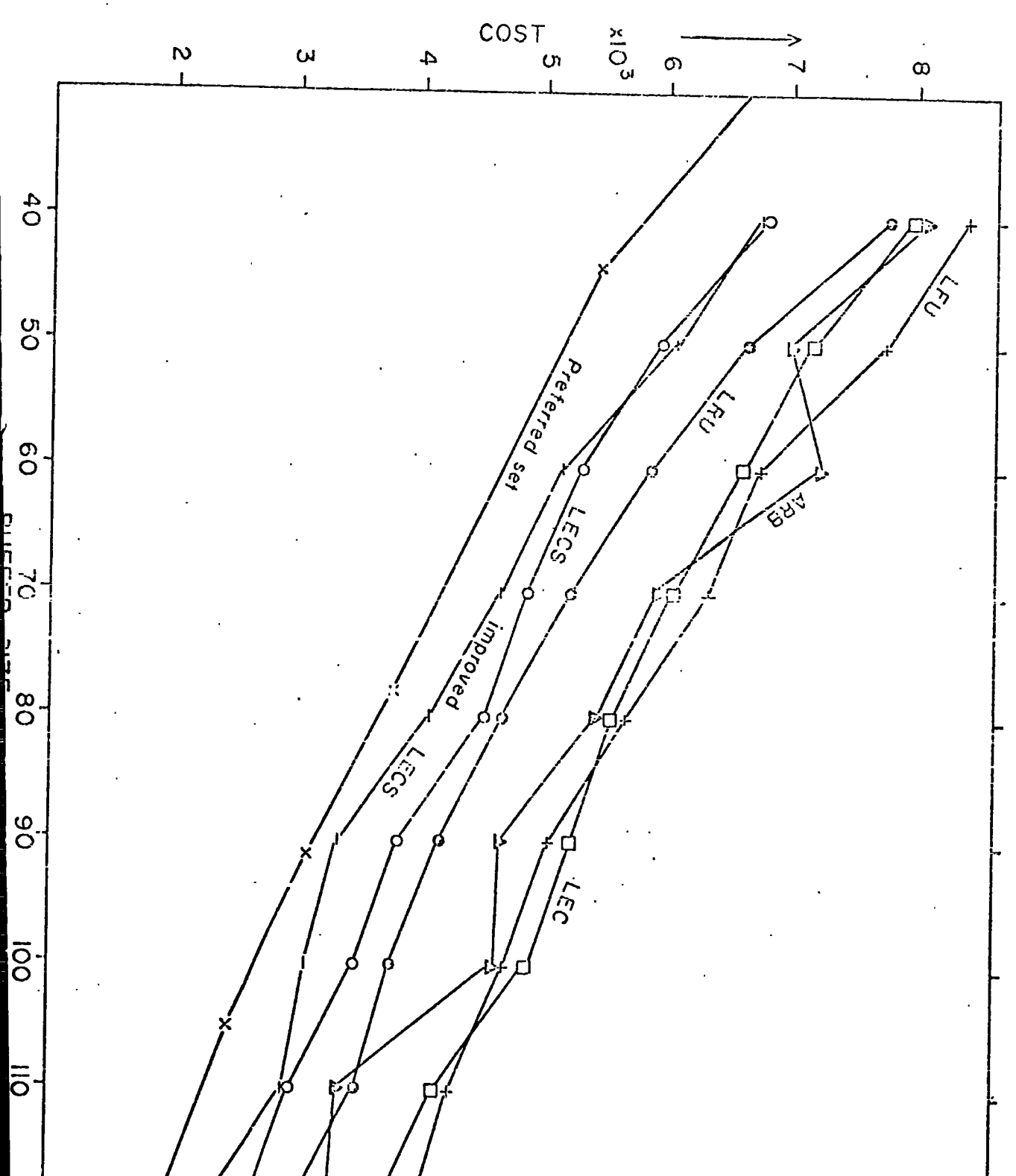


FIGURE 4.7 Cost/storage size for rapidly changing patterns

the value of α_{opt} could have been evaluated; however, no correlation was discovered.

If this model is used, suitable fixed values for α and the period should be specified. Therefore, this model is not suitable for the automatic handling of rapidly changing patterns.

2.2.2.

The second model is as follows:

At time t update the usage count of relation i as follows:

$$U_i^t = U_i^{t-1} + t$$

This gives more weight to the recent references.

This simple model proved to be satisfactory in almost all the cases.

In Figure 4.7 this model is called Improved LECS . The string used in the experiment has a working set size of 10 relations and a period of 60 references. In this experiment and others the performance of the LRU is improved when the reference pattern changes rapidly.

This model suits the strings of rapidly changing patterns. It is also simple to implement.

However, it is envisaged that if the strings of requests are very long (e.g. 1,000,000) and the counts have not been reset, the model will not be capable of giving reliable weightings, e.g. if two relations are referenced at times t_1 and t_2 which

are x references apart, the weights added due to t_1 and t_2 will practically be equal when t_1 and t_2 are very large.

In such a case the counts and the time indicator have to be scaled down, i.e. when relation j is referenced at time t:

<u>if</u> $t = T$ where T is a large number

$$\underline{\text{then}} \quad \begin{cases} u_t^i = \alpha\, u_{t-1}^i & i = 1,2,\ldots,r \\ t = 1 \\ u_t^j = u_T^j + t \end{cases}$$

<u>else if</u> $t < T$ <u>then</u> $u_t^j = u_{t-1}^j + t$

Again the choice of α and T is a problem as in 2.2.1. However, for a large T the method becomes less sensitive to the value chosen for α .

2.2.3 The maximum likelihood method

The references to each relation resemble a time varying series of events. The change in the rate of reference is used in the replacement criterion so that the relation whose usage is expected to increase is given more chance to remain explicit.

The model given by [Cox and Lewis 1966] was adopted after minor adaptation. The maximum likelihood estimate for relation r, designated G_r satisfie

$$Y(G_r) = \frac{n_r}{G_r} - \frac{n_r t_{o,r}}{1-e^{-G_r to}} + \Sigma\, t_{i,r} = 0$$

where $Y(G_r)$ is the derivative of the expression for the likelihood estimate w.r.t G_r.

n_r is the number of references to relation r excluding the last reference.

$\Sigma t_{i,r}$ is the sum of reference times excluding the last.

$t_{o,r}$ is the time of the last reference.

The above equation is solved numerically for G_r. Using Newton's method the solution required an average of 17 iterations at the first instance and subsequently, using previous results as approximations, only about 8 iterations are sufficient to find the root within a reasonable tolerance.

Each iteration requires two evaluations: one for the above equation and the other for the equation of its derivative.

Whenever relation r is referenced at time t:

$$t_{o,r} = t$$

$$t_{i,r} = u_r^t$$

$$u_r^t = u_r^t + t$$

$$n_r = f_r$$

$$f_r = f_r + 1 \text{ where } f_r \text{ is the reference count}$$

In order to choose an explicit relation to be deleted, the above equation has to be solved for each explicit relation in the above described manner.

The results are comparable with those of 2.2.2 when the string of requests has one pattern of references. As the changes become rapid its performance gets worse because the number of

references within each pattern is too small to give a good prediction. As this model is time consuming it is not recommended in its present form for this situation.

3.

The justification for the use of defined relations on a cost basis is a typical case of the disk space versus cpu time paradox. Here, two approaches are discussed:

(1) the special case of the geological data base.

(2) the optimum size of dynamic area (work space).

3.1 Defined relations in the geological data base

Consider the following statistics:

i) cost of answering 27 queries from base relations (IBM 360/44 cpu time) 2162 sec

overhead of answering a query from an explicit relation (cost of access to an explicit form) 2 sec

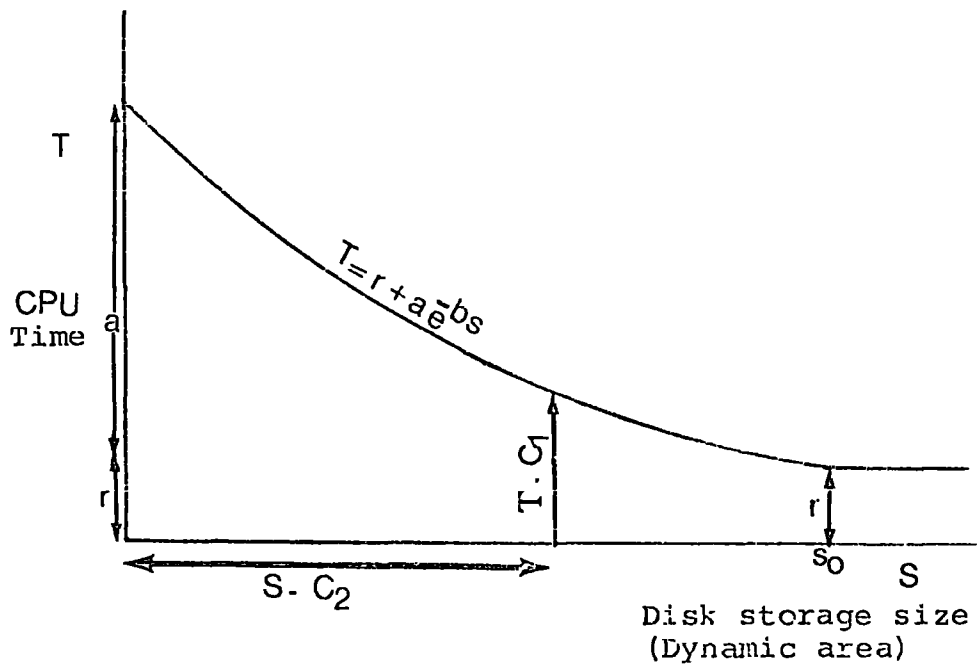
size of disk storage to hold all the 27 explicit relations: 150 blocks each of size 520 bytes 78 kb

ii) if all the relations are explicit and each relation is reference only once over a period of 1 month, then:
the gain in cpu time is 2108 sec
disk space overhead is 78 kb-month

It is evident that a high gain is achieved by trading off 11 tracks of IBM 2314 for 35 min. cpu time of the IBM 360/44. Therefore, it is fair to conclude that for cpu bound systems such as the IS/1.0, defined relations improve the performance.

3.2 The optimum size of the dynamic area

Here, the optimum disk space to be allocated for defined relations is estimated. The estimate is based on the observations of the response time and the characteristics of the relations. An expression for the optimum disk space to be allocated is presented. On the substitution of the charging rates of the particular installation, the optimum disk space is obtained and hence the retention of the currently allocated disk space can be justified or unjustified.



Consider one of the LECS cost curves of Chapter 3. For each curve the cost decays exponentially till the storage space becomes large enough to accommodate all explicit

relations ($S = S_o$).

Let the time of running a string of queries be

$$T = r + ae^{-bs}$$

where: a is the cpu time to satisfy all queries from base relations over a given period of time.

r is the cpu time to create all the defined relations for the first time.

b is a constant dependent on the replacement algorithm, the characteristics of the relations and the string of requests.

c_1 is the cost of one time unit (cpu time unit) or elapsed time unit depending on which time is used to estimate the cost).

c_2 is the retention cost of a unit of disk space for the given period of time.

[e.g. The 12 observations of the LECS curve in figure 3.13 were fitted to the above equation with a Chi-squared of 1.21, i.e. they fit with a probability of 0.995.]

From the above figure:

It is required to minimise the cost function

$$s.c_2 + T.c_1$$

substituting for T

$$s.c_2 + c_1 r + c_1 a e^{-bs} \qquad 0 \leq s \leq s_o$$

which gives

$$s_{opt} = \frac{1}{b} \log \left(\frac{bc_1 a}{c_2} \right)$$

For example:

Assuming the string of requests of the geological data base

is represented by Figure 3.13, for which $b \simeq 0.009$.

The IBM 360/44 system installed at Peterlee does not charge for the use of its facilities. In other places where this machine is installed, a flat hourly rate is employed. The NUMAC system servicing Newcastle and Durham universities has an IBM 360/67 run under the Michigan Terminal System (MTS). The MTS has charging rates specified for all the system components [MTS Users Manual 1971]. These rates were scaled down to suit the IBM 360/44.

	cpu time	Disk space 2314
MTS	.01667 units/sec	0.0000175 units/kb-hr
360/44	.00417 " "	same

For the above-mentioned 27 relations (S_o=78kb), over a time period of 10 weeks the optimum disk space is about 56kb.

4.

Here we want to estimate the probability (x) of a defined relation remaining explicit.

The dynamic storage is managed by a replacement algorithm, e.g. LECS. The values of the replacement criterion of any paged-out relation are recorded, e.g. the values of G are recorded where $G = \frac{\text{reference count} \cdot \text{cost}}{\text{size}}$. Over a period of time, knowing the number of relations paged out under the same dynamic storage size, it is possible to estimate the probability of a relation remaining explicit.

The following table explains the method of recording the values of G and the evaluation of x at a given instant of time.

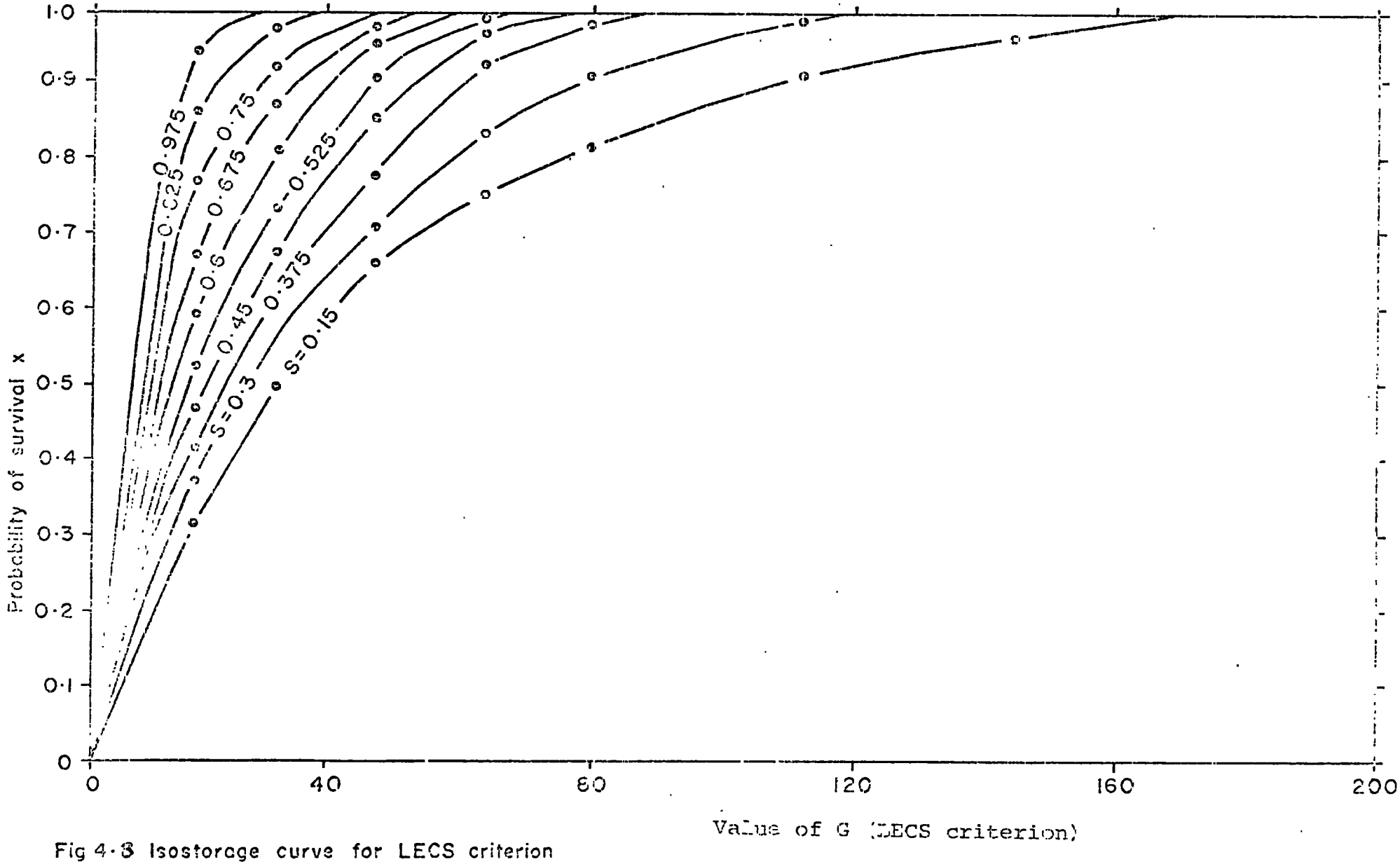


Fig 4·3 Isostorage curve for LECS criterion

	Number of relations paged out	Cumulative number of relations paged out	Probability of remaining implicit	Probability of remaining explicit x
G≥100	1	1	0.05	0.95
10≤G<100	0	1	0.05	0.95
1≤G<10	8	9	0.45	0.55
0.5≤G<1	10	19	0.95	0.05
0.1≤G<0.5	0	19	0.95	0.05
G<0.1	1	20	1.00	0.00

Now, if G can be estimated for a relation, it will be possible to predict its chances of survival in the dynamic storage.

In Figure 4.8, the probability, x, is plotted against the value of the criterion G at various storage sizes. The storage sizes are expressed as a fraction of the total storage (S) required to hold all the defined relations explicit. The probabilities are influenced by the replacement algorithm used and the type of reference string.

In Figure 4.8 a string of type (vii) (Chapter 3) is run under the LECS algorithm. This set of curves also gives the extra disk space to be added if a relation of a given G is required to remain permanently explicit. For example, at a certain instant of time when S=0.3, a relation whose G is 40 has a probability of survival, x, of 0.64. If x is required to be greater than 0.8, then S has to be about 0.525. Therefore, enough disk space should be added in order to increase S from 0.3 to 0.525.

Chapter 5

THE SPLITTING OF RELATIONS IN ACCORDANCE WITH THE USAGE PATTERN

Introduction

In a data base whose relations have a large number of domains or a large number of tuples, queries may only require a subset of the whole relation. The distribution pattern of references to the domains is usually nonuniform and therefore it is more efficient to separate the parts which are frequently referenced from those which are only occasionally referenced. This is conceptually similar to a well-known method in data processing where records are kept in two separate files: one for the moving hit group (e.g. moving customer's file) and the other for the records which are not referenced during a certain period of time (e.g. dead customer's file). Here, the case is more complex because we consider varying frequencies of reference for different groups of fields or records.

The possible ways in which a relation may be split are as follows:

A. Domainwise split

This method is suitable for relations having a large degree (e.g. relation Optics of degree 15 and relations Use and Property in the GLC data base whose degrees are 13 and 42 respectively). The domains that are frequently requested together are kept in one portion. The splitting can take place at one of the following levels:

I Physical storage level

Traditionally data base files are stored record by record. In this type of splitting the file (relation) is partitioned when the data base is reorganised. Each partition holds the fields (domains) which are requested by one query. The records (tuples) are linked by pointers or by position. In the extreme case of partitioning single fields (domains) are stored separately, in which case the file (relation) is stored field (domain) by field, i.e. the records are completely transposed. An example of this extreme case is the ROBOT system (Record Organisation Based On Transposition) [Burns 1972].

II Logical level (the splitting of relations)

In this type of partitioning the relation is broken into a smaller number of relations (obtained by projections including the key domain) in such a way that the initial relation can be regenerated by logical operations, e.g. join, union, etc.

B. Tuplewise splitting

Here the partitioning is effected on the basis of the object value of a particular domain. By selections on the object value of a domain the relation is resolved into smaller relations such that it is possible to recover the first relation by forming the union of the constituent relations.

C. Splitting by normalization

This is usually performed at the relation design stage. This type of splitting will not be discussed in this chapter. Some examples have already been given in Chapter 1.

Objectives

The objectives of this chapter are as follows:

(i) To construct a performance criterion (the gain in computing time) expressed in terms of the configuration of the relation (the number of portions, the number of domains in each portion, etc.) and the frequency of reference to each group of domains.

(ii) To examine the various ways of monitoring and recording the pattern of reference to domains and to choose a method which keeps a reasonably large amount of information within an acceptable overhead (in storage space and computing time).

(iii) To find the particular configuration of the relation (the set of subsets) which optimises the performance criterion obtained in (i) using the reference information recorded in (ii).

In the chapter, (i),(ii) and (iii) are discussed for domainwise splitting separately and then for tuplewise splitting.

The theoretical analysis, justification and proofs for the domainwise decomposition of relations at a logical level (type II above) has been thoroughly worked out [Palermo 1970, Delobel & Casey 1973, Delobel & Rissanen 1973]. However, the objectives listed here aim at improving the data base performance by splitting the relations in accordance with the way in which the user's queries reference the domains of relations. Thus the recommendations arrived at will be applicable to realistic situations.

Transparency

The partitioning recommended below will not be seen by the user. It is more convenient for the user to be familiar with a small number of relations satisfying his requirements rather than bother with a large number of parts of relations whose formations are varying.

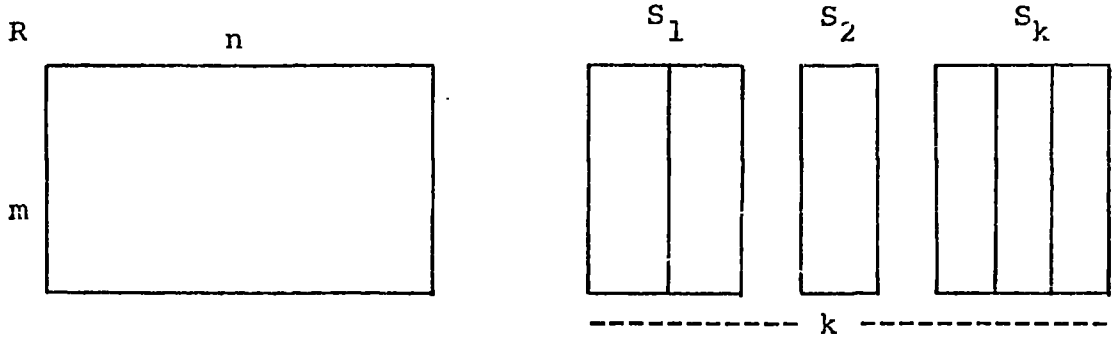
However, a clever applications programmer can achieve greater efficiency for some queries by interfacing with the data base at a lower level and choosing access paths according to the actual current states of the relations.

A. DOMAINWISE SPLITTING

The performance criterion for splitting

The domainwise splitting of relations reduces the access time if only some of the portions are referenced by queries. The optimum relation configuration is the one which gives minimum access time.

I THE SPLITTING AT PHYSICAL STORAGE LEVEL



Consider the relation R with cardinality of m and degree n. R is stored as k portions without redundancy. Portion i has a subtuple size (row size) of S_i bytes. $1 \le i \le k$.

Let the number of input buffers be f

(f is adjusted such that $f \geq k$)

the block (buffer) size in bytes be b

the disk seek time (head movement time) t_s

the disk relational delay time t_r

the data transfer time/byte t_t

the cylinder capacity in blocks C

the relation tuple size in bytes $Z = \sum_{i=1}^{k} S_i$

the proportion of non-consecutive blocks which require a seek e

Assumptions

(i) Assume that each portion of the split relation is stored in a different cylinder, i.e. a seek is required when moving from one portion to another.

(ii) Assume that the influence of the multiprogramming environment is the same if the relation is stored in either form.

Define the degree of multiprogramming interference (d) as the proportion of times the disk head leaves its former position to service a request for data not pertinent to the relation under consideration. Although it is difficult to accurately determine this parameter, it will help in drawing some conclusions later.

Consider the access time for a query requiring the reading of one portion (say portion one):

Number of blocks $= \frac{m}{b} \times S_1$

Access time per block = head movement time + average rotational delay time $\frac{t_r}{2}$ + data transfer time $(b \times t_t)$

Head movements take place:

(i) when we shift from one portion to another.

(ii) due to multiprogramming (d times)

(iii) when one portion occupies more than one cylinder (i.e. an average of $\frac{1}{C}$ movement per block), or some of the blocks are not in the same cylinder.

Head movement time/block for one portion

$$= t_s(\frac{1}{C} + d + e)$$

Access time/block for one portion

$$= t_s(\frac{1}{C} + d + e) + \frac{t_r}{2} + b \times t_t$$

Access time for one portion (portion one)

$$= \frac{m}{b} \times S_1(\frac{t_s}{C} + t_s \times d + t_s \times e + \frac{t_r}{2} + b \times t_t)$$

If the relation is not partitioned then the access time

$$= \frac{m}{b} \times Z(\frac{t_s}{C} + t_s \times d + t_s \times e + \frac{t_r}{2} + b \times t_t) \qquad \text{— (1)}$$

$$= \frac{m}{b} AZ$$

where $A = (\frac{t_s}{C} + t_s \times d + t_s \times e + \frac{t_r}{2} + b \times t_t)$

When the relation is stored as k portions and only k buffers are available (i.e. f=k) the access time to read the file (relation) record by record

$$= \frac{m}{b} \times \left(\sum_{i=1}^{k} S_i\right)(\frac{t_s}{C} + t_s \times d + t_s \times e + \frac{t_r}{2} + b \times t_t + k \times t_s)$$

$$= \frac{m}{b} \times (A + k \times t_s) \sum_{i=1}^{k} S_i \qquad \text{— (2)}$$

($k \times t_s$: the head has to jump from one cylinder to another)

However, generally when f buffers are available we assume that blocks of the k^{th} portion will be read consecutively in the f-k remaining buffers. Hence —(2) will be:

$$= \frac{m}{b} \cdot \left(\left(A \sum_{i=1}^{k} S_i + (k-1) t_s \sum_{i=1}^{k-1} S_i + \frac{S_k \cdot t_s}{(f-(k-1))} \right) \right) \qquad \text{— (3)}$$

In general, if a query requires p portions when f reading buffers are available, the access time

$$= \begin{cases} \frac{m}{b} \cdot \left(A \sum_{i=1}^{p} S_i + t_s \left((p-1) \sum_{i=1}^{p-1} S_i + \frac{S_p}{(f-(p-1))} \right) \right) & \text{— for } p>1 \\ \\ \frac{m}{b} \cdot A \cdot S_j & \text{— for } p=1 \end{cases}$$

where j is the index of the one portion requested.

Now let Q be the number of queries requiring relation R over a certain period of time.

q_{wi} be the number of queries requiring a set of portions w_i

$1 \le i \le u$ the number of sets of portions requested by queries

$u \le$ the number of possible combinations of portions (2^k-1)

$$Q = \sum_{j=1}^{u} q_{wi}$$

P_{wj} be the number of portions in set w_j

Time to answer Q queries

$$= \sum_{j=1}^{u} \begin{cases} \underline{\text{if}}\ P_{wj} > 1\ \underline{\text{then}} \\ \left(A \sum_{i=1}^{P_{wj}} S_i + t_s \left((P_{wj}-1) \sum_{i=1}^{P_{wj}-1} S_i + \frac{S_{P_{wj}}}{(f-(P_{wj}-1))} \right) \right) \cdot q_{wj} \\ \underline{\text{else}}\ A \cdot S_j \cdot q_{wi} \end{cases} \qquad \text{— (4)}$$

Example

For an IBM 2314 disk, assuming block size of 1 kbytes

$$C = 120$$

$$t_s = 60 \times 10^{-3} \text{ sec}$$

$$t_r = 12.5 \times 10^{-3} \text{ sec}$$

$$t_t = 3.3 \times 10^{-6}$$

Assume that the effect of multiprogramming is negligible,

i.e. $d = 0$

$e = 0.01$

$$A = t_s\left(\frac{1}{C} + d + e\right) + \frac{t_r}{2} + b \times t_t$$

$$A = t_s\left(\frac{1}{120} + 0.0 + 0.01\right) + \frac{t_r}{2} + 10^3 \times 3.3 \times 10^{-6}$$

$$= t_s\left(\frac{1}{120} + 0.0 + 0.01 + 0.1 + 0.05\right)$$

$$= \underline{0.17\ t_s} = .0102 \text{ sec}$$

Consider a relation whose domains are:

	A/c No.	Balance	Name	Address
object size in bytes	4	4	20	50

Suppose we have the following statistics on 1000 queries (i.e. Q=1000):

No. of queries requiring domains 1 and 2 = 800 (80%)

No. of queries requiring domains 3 and 4 = 100 (10%)

No. of queries requiring domains 1,2,3,4 = 100 (10%)

Let us work out how much gain is achieved by storing the relation as two portions:

A/c No. Balance

and

Name Address

Assume f = 3, i.e. only 3 input buffers:

$$\% \text{ saving in access time} = \frac{(\text{access time without splitting} - \text{access time with splitting})}{\text{access time without splitting}}$$

$$= \frac{A.\frac{m}{b}.Z \times 1 - A.\frac{m}{b}.(4+4) \times 0.8 + A.\frac{m}{b}.(20+50) \times 0.1 + \left(AZ + t_s\left(8+\frac{70}{2}\right)\right)\frac{m}{b} \times 0.1}{A.\frac{m}{b}.Z}$$

$$= \frac{Ax78x1 - (Ax8x0.8 + Ax70x0.1 + Ax7.8 + t_s x4.1)}{Ax78x1}$$

$$= \frac{78 - \left(6.4+7+7.8+4.1\frac{t_s}{A}\right)}{78}$$

$$= \frac{78 - \left(21.2 + \frac{0.246}{0.0102}\right)}{78}$$

$$= \frac{78-45.3}{78} = \underline{0.419} \quad (42\%)$$

Note:

(i) As the degree of multiprogramming, d, increases the saving increases (e.g. in the above example when d=1 the saving in access time will be 71 %). This is

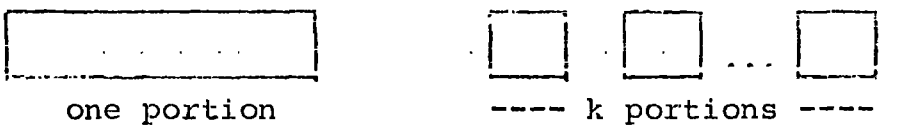
because the head will move from one portion to the other at no extra overhead.

(ii) The saving also increases with the increase of the proportion of the randomly accessed records, e.

1. Updates

Updates consist of insertions, deletions and changes of object values. Let us compare the number of portions accessed for the updates when the relation is stored with and without splitting.

No. of portions accessed



Type of update		
Change of value:		
of one object	1	1
of i values	1	varies between 1 and k
of a whole tuple	1	k
Insertion (1 tuple)	1	k
Deletion (1 tuple)	1	k

The updates do not favour a large number of portions. In general, we need k accesses per update. Now we add an update term to (4) assuming that the whole portion has to be accessed for the update.

As in (2), the update access time without splitting = $\frac{m}{b}.N.A.Z.$

the update access time with splitting = $\frac{m}{b}.N.(A+k.t_s).Z$

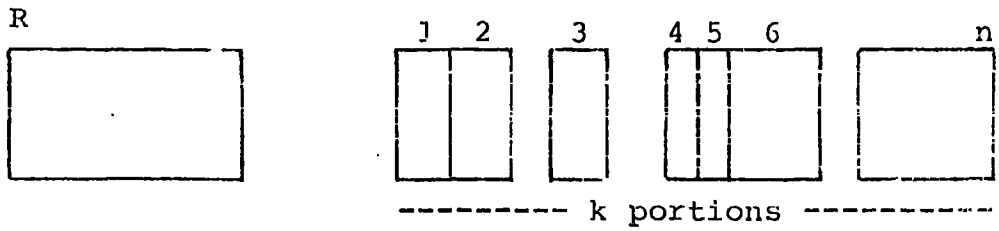
(where N is the number of updates)

Since the term for the updates is the same as that for the retrievals requiring all the k portions, we will add the number of updates to the number of retrievals requiring k portions. Hence, expression (4) will account for the retrievals as well as the updates.

2. Logical filters

In some queries the records to be retrieved are those which satisfy a logical expression (logical filter) stated as a selection criterion in the query. The logical filter may contain conjunctions or disjunctions. Each type of filter has a different influence on the access time.

As an example let us consider a relation R having n domains partitioned as shown:



2.1 Filters with the logical operator AND

e.g. to select those records for which domain(1)=a and domain(3)=b ... etc. (where domains 1 and 3 of R are in different portions).

The selection is carried out on each portion separately rather than moving from one portion to the other and hence saving in access time is achieved. If one of the terms (e.g. domain(1)=a) is false, the search will be stopped and the time to access the other portions is saved.

2.2 Filters with the logical operator OR

e.g. to select those records for which domain(1)=a or domain(3)=b (where domains 1 and 3 of R are in different portions).

In this case one of the portions has to be read $\frac{n}{f-1}$ times (where n is the number of blocks of the other portion with data satisfying the filter). This type of filter leads to loss in performance due to the increase in the access time.

The model developed so far may be extended to account for the above two cases but this approach has the following disadvantages:

(1) The model has to take into consideration the particular method of dealing with (storing) intermediate results.

(2) When (1) was included it was found that the model became complicated.

To avoid this disadvantage we will assume that the loss due to disjunctions is exactly covered by the gain from conjunctions. This assumption does not favour splitting because the number of queries having disjunction terms is usually far smaller than those having conjunction terms. This is evident in the case of the geological data base (Chapter 1, Figure 10), the analysis of the queries of the data base described by [Senko 1971] and the GLC data base (see Table 5.1).

3. The Overhead of Splitting

(i) The management of the buffers is dynamic and depends on the number of portions of the relation requested by a query. Therefore, the Read/Write routine will consume some cpu time in choosing the best buffer allocation strategy for each query.

(ii) Extra space to hold pointers showing the place of each domain in the respective portion.

(iii) When tuples are to be inserted in order to update a relation, the updating tuples have to be split before the updating of each portion takes place. Extra working disk space is required to hold each updating portion separately.

In conclusion, splitting in the manner described above is beneficial and desirable when most of the following properties of data exist:

(a) a large number of domains with an unequal distribution of reference to domains.

(b) large sizes of the less referenced domains.

(c) a relatively high number of randomly accessed records or a relatively high degree of multiprogramming interference because both of these significantly increase the access time when no splitting is performed.

II LOGICAL LEVEL (THE SPLITTING OF RELATIONS)

In this type of splitting the user's relation is partitioned by projections. For example, consider relation S whose

first domain is its original primary key or an artificial primary key (e.g. tuple number) introduced to facilitate the splitting.

The splitting

Relation S has domains $(\underline{d_1}, d_2, \ldots, d_n)$

To split S as follows:

S_1 = projection $(\underline{d_1}, d_2, \ldots d_m)$ of S

S_2 = projection $(\underline{d_1}, d_{m+1}, \ldots d_r)$ of S

S_3 = projection $(\underline{d_1}, d_{r+1}, \ldots d_n)$ of S

S_1, S_2 and S_3 are set up and stored as base relations.

Relation S is now defined in terms of its projections (S_1, S_2 and S_3).

S = equijoin on d_1 of S_1, S_2 and S_3.

1. A general approach

Generally, for queries requiring more than one relation (portion) simultaneously, portions must first be assembled into one relation equivalent to their equijoins on the primary key. This is done by submitting to the system a definition of a new relation as the equijoin of the relations (portions) requested by the query.

The defined relation will be explicit in the first place and the query will be answered. The defined relation may then be maintained explicit and can answer similar queries. This depends on the availability of space in the data base file and the characteristics of the defined relation such as its creation cost, its size and the frequency of reference to it.

For example, if a query requires domains in S_1 and S_2, relation S_{12} will be defined as:

$$S_{12} = \text{equijoin on } d_1 \text{ of } S_1 \text{ and } S_2$$

S_{12} is made explicit and the query is answered. For how long S_{12} will remain explicit depends on the aforementioned conditions.

The Overhead

Let us consider the following two extremes:

(i) An environment where the combinations of relations S_1,S_2 and S_3 which are requested by queries (e.g. S and S_{12}) can all be maintained explicit. The major overhead is only the cpu and i/o times of the equijoin for setting up each combination.

(ii) An environment where the combinations required by queries have to be set up by equijoins for each query due to scarcity of space. The overhead is the equijoin cpu time for each query.

The overhead depends on the data base environment. It is time varying and will always lie between these two extremes.

The partitioning will be beneficial if the major overhead (the equijoin cpu time) can somehow be minimised. It can be shown that splitting may possibly be beneficial only if the relations are stored with tuples sorted on some key because that will significantly reduce the join time. In the discussion that follows we are concerned with relations which are

sorted on their primary key domains. If the relation has no primary key, a system generated key such as the tuple number is added to the relation before the splitting is performed.

2. Boolean Filters

Consider a relation S having k subsets $(S_1, S_2, \ldots S_k)$ where $S, S_1, \ldots S_k$ are defined as above.

Consider the queries involving i subsets taken, for convenience, as $(S_1, S_2, \ldots S_i)$.

(a) For a boolean filter with conjunctions (AND)

Let $R = S_1$ join* S_2 join* $S_3 \ldots$ join* S_i

(where join* is an equijoin on the primary key) —(a1)

$$\text{select } (R:B_1 \& B_2 \& B_3 \ldots B_i) \equiv \text{select}(R:B_1) \text{ intersection select}(R:B_2) \ldots\ldots\ldots\ldots \text{ intersection select}(R:B_i) \quad \text{—(a2)}$$

(where B_i is a logical expression containing a domain of R which is also a domain of subset S_i.
e.g. domain(2)=5, also domain(2)=5 and domain(3)=2 if domains 2 and 3 are in the same subset.)

Substitute (a1) in (a2)

$$\text{select } (R:B_1 \& B_2 \& B_3 \ldots B_i) \equiv \text{select}(S_1:B_1) \text{ join* select}(S_2:B_2) \ldots\ldots \text{join* select}(S_i:B_i) \quad \text{—(a3)}$$

If any $\text{select}(S_j:B_j)$ =NULL then the whole expression (a3) will be equal to NULL. In other words, the selections are performed on the individual subsets and the result of the selection is equijoined on the primary key if no

result is a NULL.

Maximum number of equijoins = i-1.

In this type of query the resulting relations to be joined have a reduced cardinality. If certain conditions are satisfied it will be more beneficial to treat queries of this type, which require more than one portion, in a manner different from that stated in (1). This will be explained later.

(b) For a boolean filter with disjunctions (OR) the following expression can be obtained similarly:

$$\text{select}(R:B_1 \vee B_2 \vee B_3 \ldots B_i) \equiv \text{select}(S_1:B_1)\ \text{join*}S_2\ \text{join*}S_3 \ldots \text{join*}S_i \cup \text{select}(S_2:B_2)\ \text{join*}S_1 \ldots \text{join*}S_i \ldots \cup \text{select}(S_i:B_i)\ \text{join*}S_1 \ldots \text{join*}S_{i-1} \quad \text{—(b1)}$$

i.e. perform a selection on j^{th} subset and equijoin the result on the primary key with all other (i-1) subsets. Store the j^{th} result.
(Repeat this for j=1,...,i)
Perform a union of the i results.

Number of equijoins = i(i-1).

From the above analysis it is evident that for logical filters having disjunctive terms a large number of joins and data movements will be required. It is therefore sensible to define a relation on the subsets required by the query (by means of an equijoin) and then perform the selection on the explicit form of the defined relation.

i.e. select$(R:B_1 \vee B_2 \vee B_3 \ldots B_i)$ without splitting

$\equiv$ select$(Q:B_1 \vee B_2 \vee B_3 \ldots B_i)$

where $Q = S_1 \text{join}^* S_2 \text{join}^* S_3 \ldots \text{join}^* S_i$.

This is to say that queries having logical OR will be dealt with similarly to the general type of queries described in (1).

3. The influence of splitting on access and cpu times using the general method (1)

When a query requires more than one portion, the required portions are joined into a single relation. Since the portions are stored with tuples sorted on a certain key, the subsets are accessed block by block for the join on the equality of that key. As the join operation is performed, the query is answered at no extra access time overhead. The joined subsets are then returned back to disk as one relation so as to avoid repeating the join for repeating queries.

The access time will thus be equal to the time of accessing i portions plus the time for returning the joined i portions back to the disk as one subset. The expression for the access time is similar to that of (4).

An overhead due to the equijoin is incurred. This is equal to the cpu time of 2(i-1)m key object comparisons.

Given the expected available storage size and the characteristics of the relation to be formed, i.e. its size, cost of creation and frequency of reference, the probability of the relation remaining explicit, x, may be estimated from the isostorage survival diagram (Chapter 4).

The expected number of comparisons due to equijoins will thus be = $2(i-1)m(1-x)$.number of queries —(5)

The expected access time (cf expression 4) for i subsets

$$= \frac{m}{b}\begin{cases} \underline{\text{if}}\ P_{wj}>1\ \underline{\text{then}} \\ \left(\left[A\sum_{i=1}^{P_{wj}} S_i + t_s\left\{(P_{wj}-1)\cdot\sum_{i=1}^{P_{wj}-1} S_i + \frac{S_{P_{wj}}}{(f-(P_{wj}-1))}\right\}\right](1-x).q_{wj} \\ \quad + A\left(\sum_{i=1}^{P_{wj}} S_i\right).(1-x)q_{wj} \\ \quad + \underline{A\left(\sum_{i=1}^{P_{wj}} S_i\right).x.q_{wj}}\ \Bigg) \\ \underline{\text{else}}\ A.S_j.q_{wj} \end{cases} \quad —(6)$$

The underlined term accounts for the access time for answering the query from the single explicit relation.

S_i is the size of subset i tuple including the key domain.

The above expression accounts for the updates in the same way that has been explained before. For later reference the following special case is considered:

The access time for a relation having i subsets for one query requiring more than one subset

$$= \frac{m}{b}\left[A\sum_{j=1}^{i} S_j + t_s\left\{(i-1)\sum_{j=1}^{i-1} S_j + \frac{S_i}{(f-(i-1))}\right\}\right](1-x) + \frac{mA}{b}\sum_{j=1}^{i} S_j \quad —(7)$$

Assuming f=i

$$= \frac{m}{b}\left[(A + it_s)(1-x) + A\right]\sum_{j=1}^{i} S_j \quad —(7a)$$

4. Comparison of approaches for conjunctive boolean filters

Here is a comparison between the following two approaches for answering a query having a conjunctive boolean filter:

(i) to form a relation from the portions required by the query by means of an equijoin,

or (ii) to access each portion separately and perform a selection and then equijoin the relations resulting from the selection.

(i) The expressions for the access time and the number of comparisons for this approach have been given in (2.3) and (2.1) respectively.

(ii) Let us assume that the system follows the following steps in answering a query requiring i portions:

1. Access a portion.
2. Perform the selection on the portion and transfer the resulting relation back to the work area on disk. (Stop if the result is Null.)
3. Repeat steps 1 and 2 for all i portions.
4. Access two resulting relations. Perform the join and take the result back to disk.
5. Access the result of the last join and the next selection result. Perform the join and take the result back to disk.
6. Repeat step 5 until one resulting relation is formed.

Let us suppose that mr_j tuples of portion j satisfy the boolean filter $0 \leq r_j \leq 1 \quad j=1,\ldots i$

i.e. r_j is the fraction of the tuples of the subset satisfying the boolean filter.

If any r_j=0, the result will be NULL.

For i subsets the total access time and the number of object comparisons is as follows:

Steps 1,2 and 3 above

Time for accessing i portions for the selection and taking the result back to work area

$$= \sum_{j=1}^{i} a_j + \sum_{j=1}^{i} r_j a_j \qquad \text{where } a_j = \frac{m}{b}.A.S_j$$

Step 4

Access time for the first two subsets

$$= r_1 a_1 + r_2 a_2$$

Assuming the object values of the key domain to be uniformly distributed, the join will reduce the number of tuples of the k-1 resulting subset by r_k when joined with the k^{th} result, and vice versa.

Access time for returning the result = $r_2 r_1 + r_1 r_2 a_2$

$$= r_1 r_2 (a_1 + a_2)$$

Number of comparisons= $r_1 m + r_2 m$

Steps 5 and 6

Access time for i-1 subsets = the access time for the relation resulting from joining subsets 1 and 2 + the time for accessing the selection result of subset 3 + ... + ...

$$= r_1 r_2 (a_1+a_2) + r_3 a_3$$
$$+ r_1 r_2 r_3 (a_1+a_2+a_3) + r_4 a_4$$
$$\cdots\cdots$$
$$+ r_1 r_2 r_{i-1} (a_1+a_2+\ldots+a_{i-1}) + r_i a_i$$

Access time for returning the result $= r_1 r_2 r_3 (a_1+a_2+a_3) +$
$r_1 r_2 r_3 r_4 (a_1+a_2+a_3+a_4) + ..$
$+ r_1 r_2 r_{i-1} (a_1+a_2+\ldots+a_{i-1})$

(the time to return the last result is not included)

For steps 4,5,6: the time for accessing i subsets for the

$$\text{equijoin} = \sum_{i=1}^{i-1}\left[\left(\prod_{n=1}^{i} r_n\right)\cdot\sum_{j=1}^{i} a_j\right] + \sum_{j=2}^{i} r_j a_j$$

the time for returning the result

$$= \sum_{i=2}^{i-1}\left[\left(\prod_{n=1}^{i} r_n\right)\cdot\sum_{j=1}^{i} a_j\right]$$

Total access time for steps 1 to 6

$$= \sum_{j=1}^{i} a_j + 2\sum_{i=2}^{i-1}\left[\left(\prod_{n=1}^{i} r_n\right)\cdot\sum_{j=1}^{i} a_j\right] + 2\sum_{j=1}^{i} r_j a_j \qquad \text{—(8)}$$

Maximum number of comparisons (for steps 5 and 6)

$$= r_1 r_2 m + r_3 m$$
$$+ r_1 r_2 r_3 m + r_4 m$$
$$\cdots\cdots$$
$$+ r_1 r_2 r_{i-1} m + r_i m$$

Total number of comparisons for equijoins (steps 1 to 6)

$$= m(r_1+r_1 r_2+r_1 r_2 r_3+\ldots+r_1 r_2 r_i)$$
$$+ m(r_1+r_2+r_3+\ldots+r_i)$$
$$= m\left(\sum_{j=1}^{i}\prod_{k=1}^{j} r_k + \sum_{j=1}^{i} r_j\right) \qquad \text{—(9)}$$

	Percentage of the Queries with boolean filters having		Average fractional decrease in cardinality due to selection	Updates
	&	or	r	
Geological Data Base	10%	7%	5×10^{-4}	5%
GLC " "	22	6	7×10^{-3}	less than 1%
Poughkeepsie* " "	30	0	---	21%

[In the geological data base, 58% of the queries required joins, most of which could have been carried out using the subset of the relation which has the join criterion.]

Table 5.1

*Described by Senko (1971)

The expressions (8) and (9) demonstrate clearly that for small values of r_j the access time and the number of comparisons for method (ii) will diminish. E.G. in the geological data base the average $r=5x10^{-4}$ (see Table 5.1).

The parameters that decide which of the two methods to use are:

(a) the probability of survival, x;

(b) the average fraction by which the number of tuples diminishes after a selection, r;

(c) the number of subsets, i.

The parameter r influences the above two expressions for the number of comparisons and the access time (8 and 9) in approximately the same manner, i.e. the sign of their rate of change with r is the same. This also applies to x with expressions 5 and 7. The method chosen will therefore be better than the other in both the number of comparisons and the access time.

Comparing the number of comparisons in methods (i) and (ii) (5 and 9)

Choose method (ii) if

$$2m(i-1)(1-x) > m\left(\frac{r(1-r^i)}{1-r} + ir\right)$$

$$x < 1 - \left[\frac{r}{2}\,\frac{i}{i-1} + \frac{1-r^i}{(i-1)(1-r)}\right] \qquad \text{—(10)}$$

For at least two of the data bases in Table 1, method (ii) should be chosen.

Simplifying (8) using the average value of r_j, i.e. $r_j=r$, we obtain:

$$= \sum_{j=1}^{i} a_j + 2 \sum_{l=2}^{i-1} r^l \cdot \sum_{j=1}^{l} a_j + 2r \sum_{j=1}^{i} a_j$$

Comparing the expression (7a) and simplifying:

Choose method (ii) if

$$\left[(A+it_s)(1-x)+A\right] \sum_{j=1}^{i} S_j > A\left\{\sum_{j=1}^{i} S_j + 2\sum_{l=2}^{i-1}\left[r^l \cdot \sum_{j=1}^{l} S_j\right] + 2r \sum_{j=1}^{i} S_j\right\}$$

$$\left[(A+it_s)(1-x)-2Ar\right] \sum_{j=1}^{i} S_j > 2A \sum_{l=2}^{i-1}\left[r^l \cdot \sum_{j=1}^{l} S_j\right]$$

$$\left[(1+i\frac{t_s}{A})(1-x)-2r\right] \sum_{j=1}^{i} S_j > 2\sum_{l=2}^{i-1}\left[r^l \cdot \sum_{j=1}^{l} S_j\right]$$

which reduces to:

$$x < 1 - \frac{2r}{1+i(\frac{t_s}{A})} - \frac{2\sum_{l=2}^{i-1}\left[r^l \cdot \sum_{j=1}^{l} S_j\right]}{\sum_{j=1}^{i} S_j}$$

It is difficult to simplify this expression without making approximations. However, in order to choose method (ii) the right hand side of the above expression should be small. This is true when:

(a) r is small

(b) $S_n \leq S_{n+1}$ $\quad n=1,2,\ldots,i-1.$

Usually $r<1$, so the smaller values of S_n will be multiplied by the longer series of r whose first few terms have more significant values because they are raised to small powers. In practice this means that after the selection operation we

pick the resulting subsets for the equijoin in the ascending order of their size. In this way we minimise the amount of data being traded back and forth between the disk and the main storage.

For example,

i=3 S_1=40, S_2=60, S_3=100

$A=0.25t_s$

r=0.1

Choose method (ii) if $x < 1 - \frac{0.2}{13} - \frac{2 \cdot 0.01 \cdot 100}{200}$

$x < 0.98$

From the experiments in chapter 3, x=0.98 is usually too high to be obtained. Therefore, choose method (ii).

r=0.65

$x < 1 - \frac{1.3}{13} - \frac{0.42(40+60)2}{200}$

Choose method (ii) if $x < 0.48$

[Suppose we accessed the subsets in the following order: S_1=100 S_2=60 S_3=40

$x < 1 - \frac{1.3}{13} - \frac{0.42x(100+60)2}{200}$

$x < 0.23$]

N.B:

1. If the degree of the influence of multiprogramming is high such that $A=0.67t_s$

   then for r=0.1 choose method (ii) if $x < 0.95$
   r=0.65 choose method (ii) if $x < 0.35$
   r=0.9 choose method (i).

2. The above values of r are very large compared to those in Table 1.

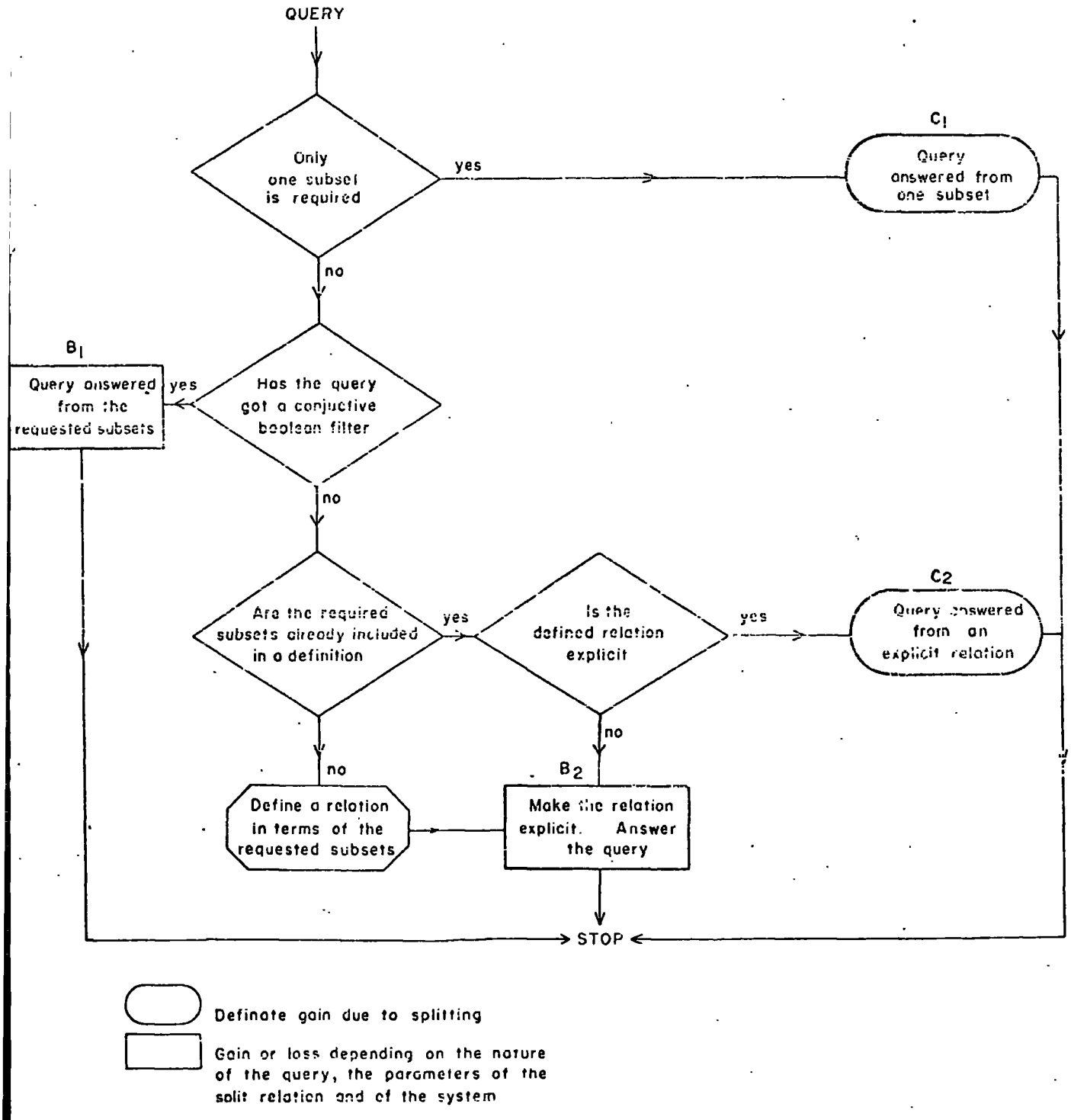

QUERY
Only one subset is required
yes
C1
Query answered from one subset
no
B1
Query answered from the requested subsets
yes
Has the query got a conjuctive boolean filter
no
Are the required subsets already included in a definition
yes
Is the defined relation explicit
yes
C2
Query answered from an explicit relation
no
no
B2
Define a relation in terms of the requested subsets
Make the relation explicit. Answer the query
STOP
Definate gain due to splitting
Gain or loss depending on the nature of the query, the parameters of the split relation and of the system

Figure 5.2

The above example shows that it is a reasonable policy to follow method (ii) for all queries having conjunctive boolean filters.

5. Summary

The flow diagram in Figure 5.2 describes an algorithm for the management of queries requiring a partitioned relation. There is a definite performance improvement due to splitting in the following cases:

1. When the query is answered from only one subset (circle C1 in the diagram).
2. When the query is answered from an explicit relation defined as the equijoin of the requested subsets (circle C2 in the diagram).

When the query requires more than one subset and the selection boolean filter is conjunctive (box B1 in the diagram), cpu time due to comparisons is traded off for gain in access time. This case does not always improve the performance. The gain in performance depends on:

a. the size of the whole relation.
b. the number and sizes of the subsets requested in the query.
c. the value of the parameter r.
d. the charging algorithm of the installation, i.e. how cpu time is weighted against the access time.
e. the judicious choice of the subsets and their constituent domains.

When the subsets have to be equijoined to answer a query (box B2 in the diagram), the loss or gain in performance depends on (a), (b), (d) and (e) above.

The frequency of each of the above mentioned four types of queries is needed in order to estimate the gain in access time due to such splitting. The expressions for the access time and the number of comparisons obtained previously for each type are multiplied by their respective frequencies. Given the cost of cpu time relative to the elapsed time, the reduction in cost can be calculated.

6. An experiment

The following example, Figure 5.3, gives the response time of queries on a relation with and without splitting. This gives a feel of the magnitude of the potential gain which can be achieved by judicious splitting.

A GLC data base relation PROPERTY

(Street#, Propp, Olduse, ..., etc.)

degree = 42

cardinality = 27323

tuple size = 124 bytes

The relation is sorted on the primary key (St#). The following portions were formed:

A = projection of PROPERTY on domains 1 and 2

B = projection of PROPERTY on domains 1 and 3

	Query	Response time (sec) IBM370/45	Relation used in answering the query
1	i) select(Street#=53&Prop=99X)	168	PROPERTY
		49	A
	ii) select(Olduse=8)	181	PROPERTY
			B The resulting relation D=select from B(Olduse=8)
2	i) select(St#=53)	46 (av.)	PROPERTY
		1.2	A The resulting relation C=select from A(St#=53)
3	i) select(St#=53&Olduse=8)	124	PROPERTY
	ii) join relation C & D on the equality of St#	5	C cardinality=125 D cardinality= 5
4	i) select(St#=53 Olduse=8)	128 (av.)	PROPERTY
	ii) join A and B on equality of St#	88	A and B The resulting relation E=equijoin of A & B on St#
	iii) select(St#=53 Olduse=8)	48	E

Results

Queries requiring more than one portion:

(a) Conjunctive queries:

Time to answer query 3(i) from PROPERTY = 124 sec.

Time to answer query 3(i) from portions by means of selections from A and B and then joining the results = 41 sec.

∴ clearcut gain

(b) Disjunctive queries:

Time to answer query 4(i) from PROPERTY = 128 sec.

Time to answer query 4(i) from portions A and B by means of equijoining on the sorted primary key domain and then answering the query from the result = 136 sec.

∴ a relatively small loss

FIGURE 5.3

III THE FORMATION OF SUBSETS FOR DOMAINWISE SPLITTING

When relation R is a candidate for splitting, some information describing the reference pattern of the domains has to be recorded. This information includes the frequency of usage and the concurrency of reference to the domains. When the data base is reorganised this information is used to evaluate the parameters required by the splitting criterion (in I and II above). Accordingly if the splitting is viable, the optimum configuration of subsets will be determined.

The information to be stored is the following:

(i) the number of times each domain was referenced as a single domain;

(ii) the number of times each particular combination of domains was requested by a query.

For a relation of n domains (i) will only require n locations while (ii) will require 2^n-1 locations (i.e. $\sum_{r=1}^{n} \frac{n!}{r!(n-r)!}$).

The latter can lead to very large storage requirements, e.g. for relation Optics, n=15, we need 32,767 locations, i.e. about 98K if the size of each location is 3 bytes. The access to such a large number of locations whenever the relation is referenced is a serious overhead.

However, the following method is recommended. A bit vector of length n is stored for every type of query. The vector has 1's in the positions corresponding to the domains involved in the query. In Figure 5.4 the first query (Q1) requires

the first and the second domains.

This bit matrix method works as follows:

(i) When a query requiring relation R is decoded the positions of the referenced domains of R are marked by 1's in a bit vector X.

i.e. X_i='1'; for all $i \in I$

where I is the set of the domains of R involved in the query

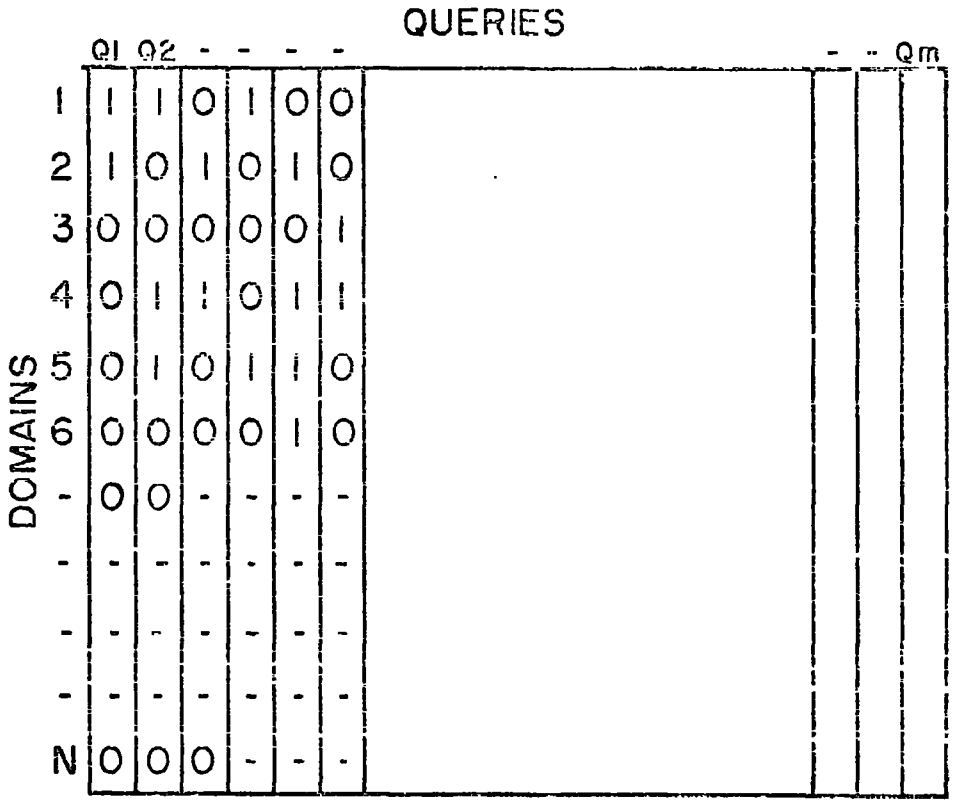
and X_j='0'; $j \notin I$.

(ii) The bit table is searched for a match between vector X and the vectors of the queries so far recorded. If no match is found vector X is added to the table to increase the number of types of query by 1.

The storage overhead of the bit table method is relatively small. It is only n bits times the number of the types of query.

The queries requiring a single domain are not added to the matrix but the respective domain reference count is updated.

Allowing for 500 query types (columns in Figure 5.4) for a relation having 16 domains a storage of 1K bytes is needed. From the observation of the usage of relations Optics (n=15) and Property (n=42) it is unlikely that the number of multi-domain query types will exceed 2n and therefore 2n is the recommended number of columns. The storage overhead is thus $\frac{n^2}{4}$ bytes.



Bit vectors representing the queries

Query 1: SELECT FROM R(domain(1)=5 & domain(2)=A);

Query 2: PROJECT(R)(1,4,5);

etc.

N.B. SELECT FROM R(domain(1)= α & domain(2)= β);
is a Query of type 1 for any value of α and β and whether the logical operator is AND or OR.
Also, PROJECT(R)(1,2); is of type 1.

FIGURE 5.4

Another overhead is the cpu time of matching a new query as in (ii) above. The condition for a match is the equality of vector X to a query type vector (a table column). The number of comparisons depends on the type of search used, e.g. the columns of the matrix may be placed in the order of the value of their content. With the columns sorted, the number of comparisons will be cut down but the columns have to be moved whenever a new type is inserted. This poses no problem because the whole matrix can be accommodated in core.

However, this is a standard problem of a table whose key is the query pattern and whose entry is the frequency of usage. The comparisons of the various methods of table updating are explained in [Knuth, vol.III].

At time=t when vector X equals column c of the matrix, the following is updated:

(1) the number of references to column c ($u_c=u_c+1$).

(2) the last reference ($L_c=t$).

(3) the recency weighted frequency ($w_c=w_c+t$).

At time=T when all the matrix columns have been used up, the new vector X should replace column p which has the least w_p of all the columns that have not been referenced during the last 2n-1 references (i.e. $L_p < T - (2n-1)$).

In this way the old reference pattern is discarded to make way for the more recent pattern.

1. The choice of the optimum set of partitions

Given the bit matrix of Figure 5.4 and the weighted usage count of each query type, it is required to find the set of subsets which optimises the performance according to the previously mentioned criterion.

At reorganisation time the bit table is converted into a matrix in which each '0' bit is replaced by a zero and each '1' bit is replaced by a one. Each row of the new matrix is multiplied by the weighted frequency of the corresponding query type. The new matrix represents the variables for a clustering process [Boyce 1968, Wishart 1969].

A modified clustering technique is employed for the analysis. In a standard clustering problem the magnitude of the values in each column is immaterial outside the same column, while in this particular application it is significant because it represents the frequency of occurrence of that type of query.

The explanation of the clustering technique used is given in Appendix 6.

The output of the clustering process gives a maximum of n possible choices of subset. Using the input matrix of Figure 5.4 the cost of answering the queries is evaluated for each solution set. The set having the minimum cost is chosen. One of the possible solutions is the choice of a single 'subset' with all the domains. This indicates that splitting is not recommended.

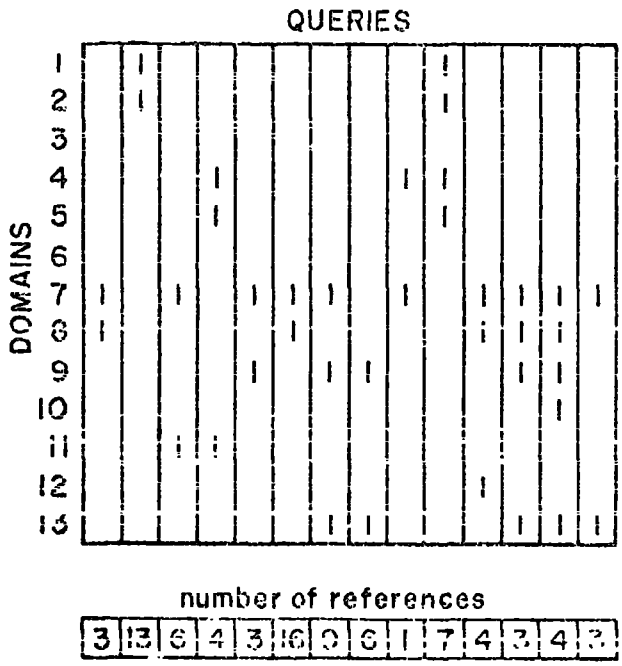
A detailed example of the choice of the partitions follows.

Example:

Relation name : USE

degree =13

cardinality = 64501



Number of queries examined = 136

Number of multi-domain queries = 82

Number of single-domain queries = 54

THE OUTPUT OF THE CLUSTERING PROGRAM WAS AS FOLLOWS:

No. of partitions	Partitions
13	(1) (2) (3) (4) (5) (6) (7) (8) (9) (10) (11) (12) (13)
2	(3,6) (1,2,3,6,7,8,9,10,11,12,13)
3	(3,6) (4,5) (1,2,6,7,8,9,11,12,13)
3	(3,6,10) (4,5) (1,2,7,8,9,11,12,13)
2	(3,4,5,6,10) (1,2,7,8,9,11,12,13)
3	(3,4,5,6,10) (1,2) (7,8,9,11,12,13)
4	(3,4,5,6,10,12) (1,2) (9,13) (7,8,11)
4	(3,4,5,6,10,12) (1,2,11) (9,13) (7,8)
3	(3,4,5,6,9,10,12,13) (1,2,11) (7,8)
2	(1,2,3,4,5,6,9,10,11,12,13) (7,8)
1	(1,2,3,4,5,6,7,8,9,10,11,12,13)

This is different from standard clustering in that in the latter the rightmost subset is considered unclustered. Though the domains in such a subset do not possess a strong similarity, they originally belong to one relation. Also, the first set of single domains does not appear in a standard clustering problem.

Now, the program chooses the solution set which optimises the splitting criterion.

Optimum set: (3,4,5,6,10,12) (1,2) (7,8,9,11,13)

The clustering operations are normally time consuming but in this application the number of characters and forms (see Appendix 6) are small compared with an average taxonomic problem. Figure 5.5 shows the cpu time taken to cluster and choose the optimum splitting set for relations of different degrees. It should be noted that this will be the actual overhead incurred by a data base system at the time of reorganisation because exactly the same process will be followed.

It is therefore appropriate to conclude that the overhead cost of splitting is small compared with the costs related to processing relations of large degrees. And that the splitting can be performed automatically at a tolerable overhead.

No. of Domains	No. of Query Types	Fortran Program cpu time(sec) IBM 360/67	Virtual Memory Kb
13	14	$\simeq 1$	less than 142
42	84	12	142
100	200	167	180

The overhead of choosing the subsets of a relation

Figure 5.5

B. TUPLEWISE SPLITTING

A relation R is split according to the value of a given domain. The resulting portions are themselves relations. These portions are formed by selections specifying a certain value or a range of values of the splitting domain. Relation R is seen as the union of these portions and the relation, R_{rem}, formed by the remaining tuples which do not satisfy the selections.

i.e. for a relation of n portions

$$R = \bigcup_{j=1}^{n} P_j \qquad \text{where } P_n = R_{rem}$$

This splitting has the following advantages:

(i) It reduces the cpu and the access times for queries requiring the splitting domain or selections with conjunctive boolean filters containing the splitting domain,

e.g. in Example 1

```
SELECT FROM CHEMISTRY(OXIDE=FE2O3&QUANTITY>10);
```

(ii) As shown in Example 1, it is possible to save storage space by removing the objects of the splitting domain in the portions. This can be done if the relation is split on specific values of the splitting domain rather than a range of values.

The disadvantages of tuplewise splitting are the following:

(i) Each update requires some comparisons (of the order $n\log_2 n$ or $n/2$ comparisons depending on the organisation

of the portions index). Updates may also need to access more than one portion depending on the values of the splitting domain and of the updating tuples.

(ii) Queries requiring the other domains or selections having disjunctive boolean filters containing the splitting domain will need to access more than one portion. This increases the access time.

1. The levels in the hierarchy

As seen in Example 1, it is possible to further split the portions till the number of levels in the hierarchy is equal to the number of domains of the relation. It will, however, be difficult to update the resulting tree of relations. Since we are dealing with relations and not records or objects, the management of the avalanche of the resulting relations is tedious. The use of the union and the other relational operators will be exhausting.

Therefore, in the context of relations, this splitting should be thought of as forming a one-level hierarchy of relations. If, however, after reorganising the data base the portions are stored as base relations and relation R is defined as their union, then each of the portions may be further split as an independent relation.

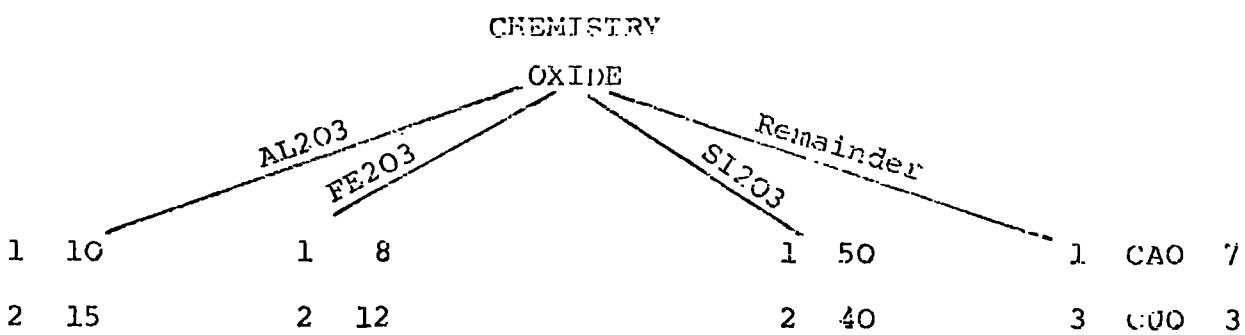
On the other hand, if each portion is further split on the same splitting domain, a multi-level hierarchy will be formed. This type of splitting narrows the range of the value of the splitting domain covered by the portion, as will be discussed later.

Example 1

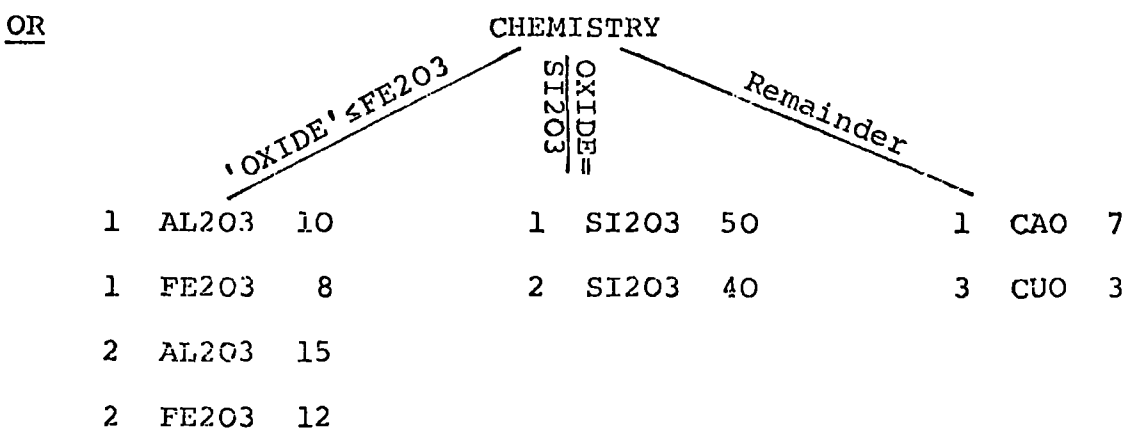
Chemistry

Ref #	Oxide	Quantity
1	AL2O3	10
1	SI2O3	50
1	FE2O3	8
1	CAO	7
2	AL2O3	15
2	FE2O3	12
2	SI2O3	40
3	CUO	3

Relation Chemistry split on domain Oxide



OR



2. The profitability of the method

Let us follow a simplified example from which we can draw some inferences regarding the profitability of the method.

Assume a relation R is split into i equal portions.

Let the number of tuples of the whole relation be M

the cost of the cpu time per object comparison be 1 unit

the cost of access time for moving from one portion to the other be W units

the frequency of queries and updates of j subsets f_j

$$\left(\sum_{j=1}^{i} f_j = 1 \right)$$

The cost of processing a k-domain selection before splitting

$$= kM + W \qquad (1 \leq k \leq \text{degree of R})$$

After splitting:

Queries on domains other than the splitting domain will require all the i portions and will cost

$$f_i(kM + iW)$$

Queries on the splitting domain and selections having conjunctive boolean filters containing the splitting domain require one or more portions

$$f_j \times j \times \left(\frac{kM}{i} + W\right) \qquad j=1,\ldots,i-1$$

$\therefore$ Cost with splitting $= \left(\frac{kM}{i} + W\right) . \sum_{j=1}^{i} j.f_j$

for the splitting to be profitable

$$\left(\frac{kM}{i} + W\right) . \sum_{j=1}^{i} j.f_j < (kM + W)$$

If we assume that all queries on the splitting domain will only require one of the subsets, i.e. $f_k=0 \quad k=2,\ldots,i-1$; the above expression reduces to

$$(\frac{kM}{i} + W)(f_1 + i.f_i) < (kM + W)$$

$$\text{i.e. } (\frac{kM}{i} + W)(f_1 + i(1-f_1)) < (kM + W)$$

$$\text{which simplifies to } f_1 > \frac{1}{(\frac{kM}{W})\frac{1 - \frac{1}{i}}{i-1} + 1}$$

where f_1 is the frequency of queries requiring the splitting domain

f_i is the frequency of all the other queries

The improvement in performance increases with the number of tuples of the relation. The increase in i improves the performance if the frequency of queries satisfied by a smaller number of portions increases proportionally.

Tuplewise splitting trades off the access time for the cpu time. It is ideal for a cpu bound system where the tuples of relations are stored unsorted, e.g. in the geological data base the splitting of relation CHEMISTRY into ten portions improved the performance significantly. In Example 2 the results of splitting into four portions are reported. If the tuples of the relation have already been stored sorted to the splitting domain, the gain in cpu time will diminish.

The penalty for queries requiring all the domains is relatively low because the union is usually less costly.

Example 2

A subset of relation CHEMISTRY was split on the second domain (OXIDE) into 4 portions:

1. ALUMINIUM
2. FERROUS
3. SILICON
4. OTHERS

An application program managed the access to the 4 portions by converting the syntax of the queries to suit the new set-up.

	BEFORE SPLITTING		AFTER SPLITTING	
	QUERY	TIME	QUERY	AVERAGE TIME
1.	LOAD(CHEMISTRY); SELECT(OXIDE=AL2O3);	39 units	LOAD(ALUMINIUM);	2 units
2.	LOAD(CHEMISTRY); SELECT(OXIDE=SI2O3) &QUANTITY>40);	60 units	LOAD(SILICON); SELECT(QUANTITY>10);	12 units
3.	LOAD(CHEMISTRY); SELECT(Ref#=212);	39 units	LOAD(ALUMINIUM); LOAD(FERROUS); UNITE; LOAD(SILICON); UNITE; LOAD(OTHERS); UNITE; SELECT(Ref#=212);	54 units
4.	LOAD(CHEMISTRY); SELECT(Ref#=212& OXIDE=AL2O3);	58 units	LOAD(ALUMINIUM); SELECT(Ref#=212);	11 units

In this system of queries of type 1,2 and 4 account for over 25% of the queries, the splitting will be justified. The frequencies of reference to relation CHEMISTRY in Figure 1.6 justify the splitting.

3. The choice of the splitting domain

A novice user will be embarrassed if he is asked to indicate the possible splitting domains or the domain involved in the majority of his queries. The choice of the splitting domain should therefore be automatic and is made when the data base is reorganised. The frequencies of reference to the domain alone and in conjunctive boolean filters are kept for each domain. If the domain with the highest sum of frequencies satisfies the splitting criterion (a generalisation of the forementioned one), the splitting is performed.

The references are weighted for the recency of usage, i.e. a reference to domain k at time=t

$$u_k = u_k + t$$

where u_k is the weighted reference for domain k.

4. The formation of portions

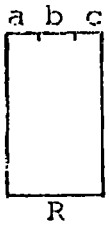
Given the splitting domain there are two possible methods for splitting the relation:

4.1 The static method

(a) The splitting domain is scanned and the range between its lowest and the highest object values is divided by the number of the required portions to give the limits of the ranges of the value for each portion. The resulting portions will be unequal in size.

(b) The reference to the portions is monitored and accordingly portions are further split as in (a)

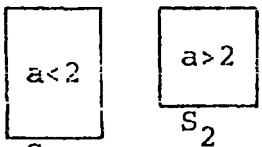
DEFINITIONS QUERIES



(a=splitting domain)

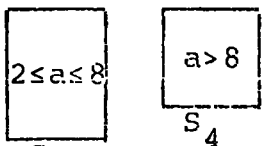
Q1: select(a>2);

$R=S_1 \cup S_2$



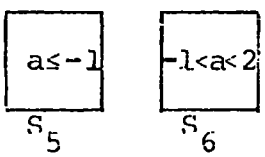
Q2: select(a>8&b<5);

$S_2=S_3 \cup S_4$



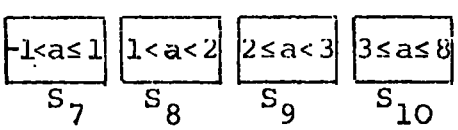
Q3: select(c=10);
Q4: select(a<-1);

$S_1=S_5 \cup S_6$

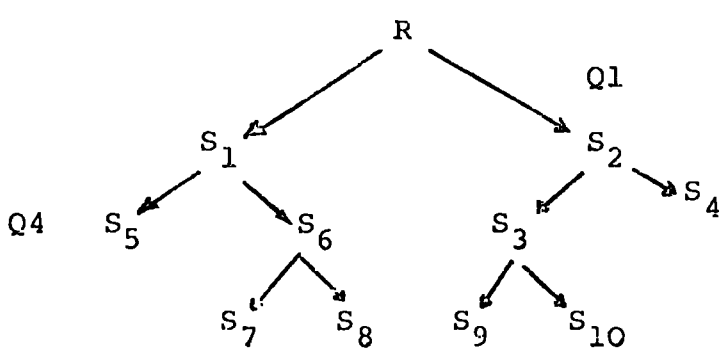


Q5: select(a=3ora=0.5);
Q6: select(a>1&a<3);
[answered by $S_8 \cup S_9$]

$S_6=S_7 \cup S_8$
$S_3=S_9 \cup S_{10}$



Q7: select(a>8&c=4);



A directed graph with no closed loops. At most one branch enters a node.

Note: Relations $S_5, S_7, S_8, S_9, S_{10}$ and S_4 are now base relations (i.e. the terminal nodes are base relations).
Relations R, S_1, S_2, S_6 and S_3 are defined split relations.

Figure 5.6

or they are united together such that the references are more uniformly distributed among them.

The set-up time is the major overhead. The performance is not very sensitive to non-uniformly distributed frequency of reference to portions. Though the portion having a very low frequency of reference degrades the performance by increasing the access time, yet it prevents the waste of cpu time by not being scanned more frequently.

4.2 The dynamic method

As in Figure 5.6, a selection on domain 'a' is satisfied by splitting R into two portions; one providing the answer for the query and the other has the remaining tuples. Relation R is defined as the union of the two portions. A selection leads to a split of a portion if its boolean filter contains the splitting domain only or if it contains the splitting domain in a conjunctive expression. In both cases the value range specified for the selection should be continuous.

The weighted frequency of reference for each portion is kept. When the number of portions exceeds a specified number, some of the portions are pruned and the others are allowed to grow (split) further. The pair of leaves (portions) having the smallest frequency of reference is united and the definition of its predecessor is deleted.

e.g. In Figure 5.6 if S_7 and S_8 are to be pruned, S_6 is made a base relation and its definition is deleted.

The dynamic method has the advantage of following the pattern of references. No major overhead is incurred when portions are set up because one of each pair answers a query. Another advantage is that since all the generated definitions contain union operations only, complicated queries do not cause complications, e.g. A join R will give

$$A \text{ join } S_1 \cup A \text{ join } S_2 \text{ etc.}$$

It is however cumbersome to maintain the portions and to program the management of pruning and splitting of portions. The tree will permanently be influenced by the first few splits and so some queries will continue to be answered by more than one portion irrespective of their frequency, e.g. (Q7 in the example) because only one branch can enter a node. This disadvantage is removed by reorganisation.

5. Comparison between the tuplewise splitting and the indexed sequential organisation

The organisation of the portions of the relation described above bears some similarities to the indexed sequential organisation. Since the latter is a well-established method, it will be useful for the implementer of the above recommendations to borrow some of its techniques and to learn from it.

Tuplewise splitting of relations	Indexed sequential organisation
A method for 'organising' relations at logical level.	A method for the organisation of files at physical level.
A union of the portions is formed when queries require	Processes a file serially with the advantage of random

Tuplewise splitting of relations	Indexed sequential organisation
domains other than the splitting domain. (The union should not necessarily involve the physical linking of one portion with another.) For other queries only a subset of the portions is scanned.	access to skip inactive records.
It has an associated index which maps object values onto relations. The index may be built automatically.	It has an associated index (or access file) which maps key values onto location or (disk) addresses. Such indexes exist at each level of the hierarchy.
May be reformed at the user level.	Usually reformed at the system level.
The overhead due to updates increases after splitting. The index is not affected by the updates.	The file is altered and copied for each update. The index is also affected by the updates. However, with slight modification to the index technique, insertions can be held in an 'overflow' area assigned to the cylinder, thus avoiding the copying after updates.

Tuplewise splitting of relations	Indexed sequential organisation
Needs periodic reorganisation to redistribute the value ranges in accordance with the reference pattern.	Needs periodic reorganisation to restore it into a serial file.

6. Final remark

The improvement in the performance of tuplewise splitting will diminish to a varying extent if:

(a) the distribution of the queries among the most frequently referenced domains is nearly uniform,

(b) the pattern of reference is very rapidly changing.

(c) the tuples are sorted on the most frequently referenced domain.

A special indexing technique is appropriate for the above three cases and is described in the next chapter.

Chapter 6

DEFINED RELATIONS AS INDEXES

Introduction

The use of index files for accessing records, on the basis of secondary key values, is a common feature in information systems. The advantage of indexing is the improvement of the response time. If a domain within a relation were to have an associated index, then an object within that domain can be selected without requiring a serial search through the domain. The use of indexes for reducing the join operation time has been discussed by [Palermo 1972].

The selection of the key domains (fields) is of great importance and is a major factor that leads to the improvement in the performance. Two possible ways to approach the problem of setting up indexes for a relation are:

(a) Form indexes for all domains in a relation in anticipation of possible queries. This has the disadvantage of wasted storage space and processor time for indexes which may never be used.

(b) Form indexes for only those domains that are frequently used as search keys. Bearing in mind that a constantly changing pattern of queries would require the destruction and recreation of indexes, this method leads to better secondary storage utilization but

problems in monitoring.

In this chapter an attempt is made for solving this problem within the scope of relational data bases by employing the defined relations capability. The discussion consists of:

(1) the defined relations as indexes and the results of experiments which support this approach.

(2) a detailed consideration of the choice of the domains to be indexed and the stage at which the indexing should be done.

(3) indexing as a form of relation splitting.

1.1 The defined relations as indexes

For relation R of degree n and cardinality m define relations R_i, i.e.(i=1,2,3,...,n)
Each of these n relations is defined on K as

R_i = FORMINDEX (i,R);

In IS/1.0 this is as follows:

```
DEFINE (*1*R); /*DEFINE RELATION R1*/
LOAD (R);
FORM (1); /*FORMS AN INDEX ON DOMAIN 1 */
END;
```

The application program (FORM) which forms the index does the following (see Figure 6.1):

(a) It forms a binary relation, the first domain being the domain of R which is going to be sorted. The second domain is a pointer to a position in R (or a tuple number of R).

R	1	2	.....n
1	800	HASWELL	
2	50	SHERBURN	
3	1	NEWCASTLE	
4	27	DURHAM	
5	40	PETERLEE	

Before sort

key	pointer (tuple#)
800	1
50	2
1	3
27	4
40	5

(a)

After sort

Domain 1 of R

R_1

Domain 1 of R	Pointer (tuple#)
1	3
27	4
40	5
50	2
800	1

b(i)

R_1

Pointer (tuple#)
3
4
5
2
1

The tuple number of the {smallest / largest} object value in domain 1

b(ii)

In the implementation $R_1, R_2, \ldots$ are referred to as *1*R, *2*R, ...

Similarly :

After sort

Domain 2 of R

R_2

Domain 2 of R	Pointer (tuple#)
Durham	4
Haswell	1
Newcastle	3
Peterlee	5
Sherburn	2

OR

Pointer (tuple#)
4
1
3
5
2

FIGURE 6.1

(b) It sorts the binary relation on the key domain.

Two versions are available:

i) the defined index is a binary relation of a key and a pointer (Figure 6.1 b(i)). When selecting a value the binary index is searched by a binary chop and the result is a set of keys (or null) and their pointers.

ii) a unary relation is formed (Figure 6.1 b(ii)). It is a list of pointers to the objects of the source domain which have got a similar position when sorted. This type saves the storage of the key domain and is useful for domains of large size, e.g. 50 bytes domains of relations Reference and Geography. It consumes access time in referring back to relation R for each object comparison.

Sorting:

The sorting methods employed were the Quicksort [Rich 1972] and the two way merge adapted to internal sorting. The former is usually faster. The latter is faster for semi-sorted strings of data such as the Ref# in the geological data base.

Updates:

The update performance after indexing is made up of two parts:

i) improvement in updating the main relation R in the cases of deletion and value changes.

Relation : Standing Orders or S
(A/C#, Name, Amount, EXPIRYDATE)
Indexed on A/C#, Name

Type of Update	Improvement in the update performance of relation R	Overhead of updating the indexes incurred ONLY when an index is explicit
A. INSERTION	same as without indexing	1. processing time equivalent to the time of a selection ($2+\log_2 m$) for finding the position of the insertion. 2. the copying or the reorganisation of the indexes to accomodate the inserted entries.
B. DELETION	1. Using an indexed domain key e.g. delete from S the tuples for which (A/C#=743201). The deletion is faster with indexing. The number of comparison drops from an average of m/2 to $2+\log_2 m$.	1. processing time to locate the entry to be deleted (as A(1) above. 2. either the recopying or the reorganisation of the indexes for each group of deletions, OR the introduction of void locations and periodic reorganisation.
	2. Using a non-indexed domain key e.g. delete from S the tuples for which (EXPIRYDATE=74.05) Same as without indexing.	B(2) above
C. CHANGE OF VALUE	1. Using an indexed domain key to change an unindexed domain e.g. alter the expiry date for (A/C#=743201) As B(1) above.	No overhead because Expirydate is not indexed
	2. Using an indexed domain key to change the same domain or another indexed domain e.g. change AC# of (Name=OTHER) to (AC#=55367). The change of value is faster with indexing (as in the case of deletion).	1. locate the entry to be changed (as B(1) above). Either delete the entry or flag it. 2. insert the new entry (as A(1) above. 3. reorganise index (as A(2) above).

FIGURE 6.2 The influence of the update performance by indexing

ii) overhead due to updating indexes. This is incurred only when an index is explicit. The number of the explicit indexes affected by the update depends on the update type and the domains involved.

The table in Figure 6.2 discusses the overhead due to each type of update.

1.2 Experiments

In this section I intend to justify the use of defined indexes by showing the improvement in the overall performance of the data base brought about by such indexing.

The data base file had a Regional(1) organisation. Regional(1) is a PL/1 file organisation which allows the file to have direct or sequential access as well as input or update [(PL/1(F) Language Reference Manual].

Within each relation the access to the tuples was designed to be serial. For example, if tuple 1000 is requested when tuple 10 is in the buffer, tuples 11 and 999 have to be passed before tuple 100 is accessed. The overall gain would have been many times higher if the design was made bearing in mind the possibility of such indexing. However, that was not one of the design objectives of the early version of the experimental system.

The high read/write overhead of the experimental system has made the time of the sort, FIND and SELECT high. In spite of this the experiments still give an indication of the power of the method of defined indexes.

The FIND operator

This is a PL/1 written routine which is interfaced with the system via the system macros. It carries out the function of the select operator using indexes.

It works as follows:

e.g. LOAD(R);

FIND(TOWN=PETERLEE);

i) it decodes the filter and initializes the search parameters.

ii) it requests relation named

*<the number of the domain whose name is 'TOWN'>*R

e.g. *2*R

iii) using the index relation and relation R, it performs a binary chop and outputs the result.

Comparison of SELECT and FIND times

In the experimental system the cardinality of the relation to be selected from should not exceed an arbitrary value of (3200). Therefore, the relations GEOGRAPHY and MINERALNAME have been chosen for the comparison between the SELECT and the FIND operators.

Figures 6.3a and 6.3b show that if an index is created and referenced a certain number of times an improvement in the response time is achieved. Figure 6.4 illustrates the case when the index is made implicit before a sufficient number of references is made to it. Before it recovers the overhead of a sort another sort becomes imminent. In such a case indexing degrades the performance.

Relation GEOGRAPHY

cardinality = 2208

degree = 4

	CPU time (units)		CPU time (units)
1. A general selection			
LOAD(GEOGRAPHY);		LOAD(GEOGRAPHY);	
SELECT(REF#=888);	114.7	FIND(REF#=888);	66.4
LIST;		LIST;	
2. The required object is smaller than the smallest object in the domain			
LOAD(GEOGRAPHY);		LOAD(GEOGRAPHY);	
SELECT(REF#=1);	101.8	FIND(REF#=1);	51.1
LIST;		LIST;	
3. The required object is greater than the largest object in the domain			
LOAD(GEOGRAPHY);		LOAD(GEOGRAPHY);	
SELECT(REF#>3000);	113.7	FIND(REF#>3000);	54.1
LIST;		LIST;	
		The time to create the index (includes the sort time)	
		LOAD(*1*GEOGRAPHY);	90*
		LIST;	

*Nett time without overheads

FIGURE 6.3a

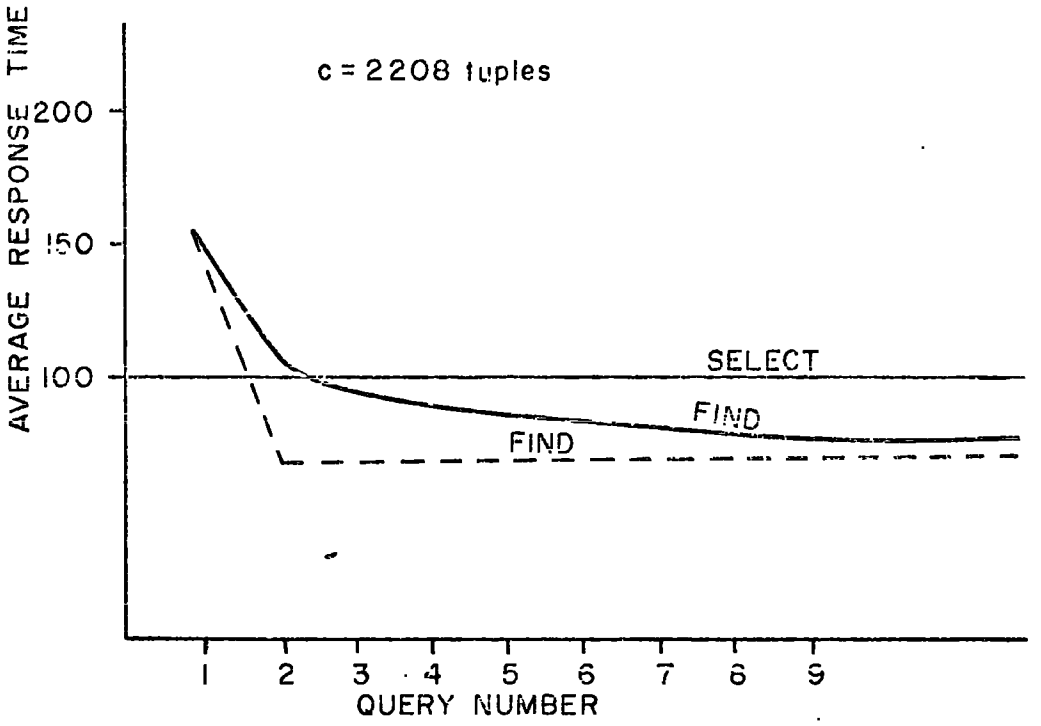
Relation MINERALNAME

cardinality = 2444

degree = 2

	CPU time (units)		CPU time (units)
1. A general selection			
LOAD(MINERALNAME); SELECT(NAME='FERRO KAERSUTITE'); LIST;	144.5	LOAD(MINERALNAME); FIND(NAME='FERRO KAERSUTITE'); LIST;	78
2. Object value too large (case 3 above)			
LOAD(MINERALNAME); SELECT(NAME=XENOTIME); LIST;	128.8	LOAD(MINERALNAME); FIND(NAME=XENOTIME); LIST;	60.5
3. Conjunctive filter and the result			
LOAD(MINERALNAME); SELECT(REF#>2500 & NAME=HORNBLENDE); LIST;	256.4	LOAD(MINERALNAME); #FIND(REF#>2500 & NAME=HORNBLENDE); LIST;	92
4. Disjunctive filter			
LOAD(MINERALNAME); SELECT(REF#=200 | REF#=5) LIST;	228.4	LOAD(MINERALNAME); FIND(REF#=200 | REF#=5) LIST;	106
		The time to create the index LOAD(*2*MINERALNAME); LIST;	100

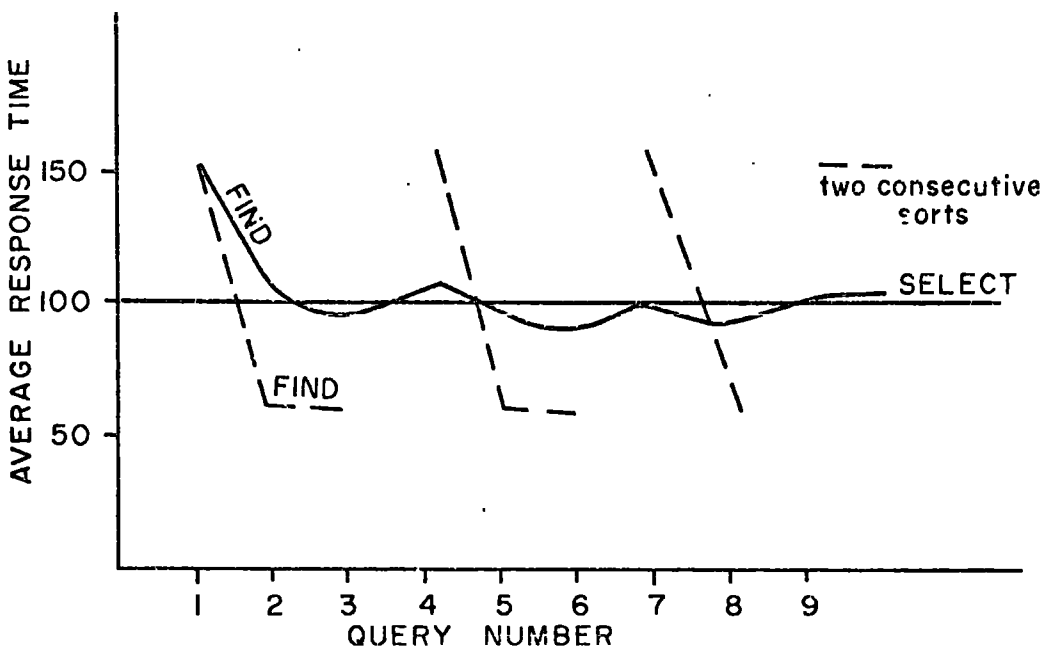
FIGURE 6.3b



(a) Example of a case where indexing improves the performance

— — — Response time

$$\text{Average response time} = \frac{\text{Cumulative response time}}{\text{Number of queries}}$$



(b) Example of a case where indexing degrades the performance

Fig 6·4

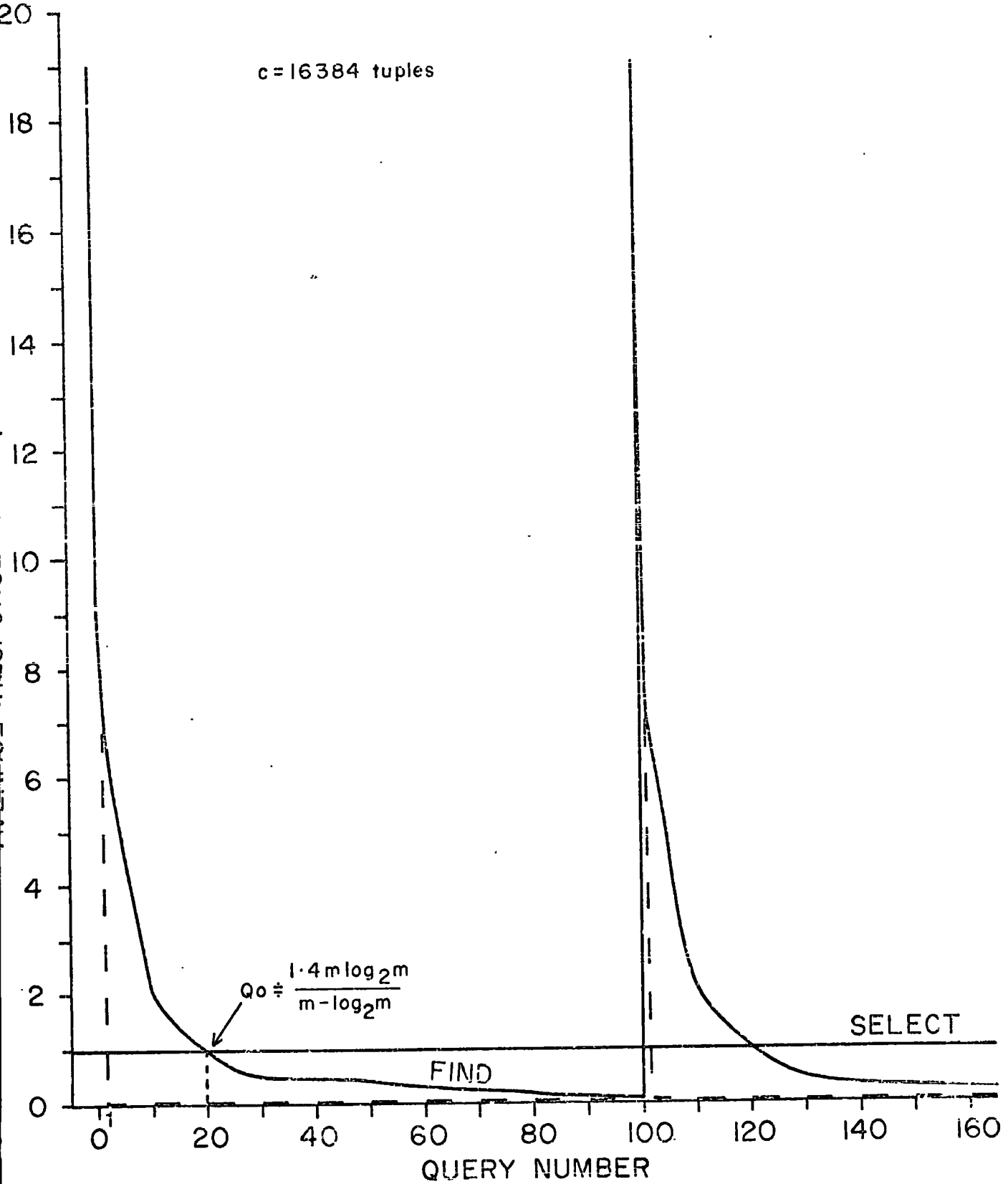


Fig 6·5 Effect of Indexing on performance (Theoretical)

Some other functions were adapted to make use of the indexes. This reduces the times of these functions and lowers the overall average response time. Examples of these functions are the IS/1.0 MINIMUM, MAXIMUM, REMOVE, INTERSECT and DIFFER.

```
e.g.  LOAD(R);
      MINIMUM(2); /*FINDS the smallest object in domain 2.
                    Calls index *2R */
      LOAD(R);
      MAXIMUM(2);   etc.
```

However, the results of the above experiments do not fully justify the use of such indexes because the disk storage space has not been taken into account and because the results obtained are influenced by the characteristics of the implementation.

However, Figure 6.5 uses the theoretical values to depict the effect of defined indexes on retrieval.

2.1 A criterion for efficient indexing

As will be explained later, the efficiency of indexing depends on:

(a) the references to the index
(b) the type of references whether updates or retrievals.
(c) the storage available in the system.
(d) the number of tuples in the relation.

To find a criterion for choosing the domain to have an associated index, let us consider the following:

i) At any instance given the frequency of reference to the prospective index, its expected size (depends on object size and cardinality) and its creation cost (proportional to $m \log_2 m$: where m is the cardinality); then it is possible to estimate the probability of the survival of the index in the explicit form, x, using the isostorage diagram (Chapter 4).

To estimate the cost of the computer time for the creation and maintenance of indexes we need to know the cost of the basic components, i.e. the number of the disk accesses and the number of object comparisons. Here, the cost of computer time is assumed to be proportional to the number of comparisons because:

(a) the number of disk accesses is related to the number of comparisons in the significant operations, e.g. the sort and the selection.

(b) it is simpler to estimate the number of comparisons.

ii) Number of comparisons for a one domain selection if the domain is not sorted is:

$$m$$

For each selection from an indexed domain

$$2 + \log_2 m \text{ comparisons are required}$$

Whenever the index is found implicit, it has to be sorted. The expected number of comparisons =

$$(1.386 m \log_2 m)(1-x) \text{ [Gerhard 1974]}$$

where (1-x) is the probability of the relation being implicit.

iii) the updates:

The example in Figure 6.2 shows that:

(a) an update consists of two processes: locating the tuple and updating part or the whole tuple.

(b) even if the cost of disk storage is negligible and infinite space is available, an index may still degrade the performance (response time) when it is updated. In the example, the index on domain Name has to be updated for every insertion or deletion and for some value changes.

The latter point illustrates the overhead of keeping an index which is not frequently used for locating tuples. It is therefore important to have two reference counts for each index:

LOC the number of times the index was used to locate a tuple or a group of tuples for retrieval or update;

and MOD the number of times the index was modified.

A domain index that does not improve the processing time is an overhead and should be destroyed, i.e. an index of a domain must be destroyed if over a period of time the expected number of comparisons without having that index ≤ the number of comparisons when the domain has an index.

i.e. $LOC \cdot m \leq (LOC + MOD) \cdot Log_2 m$

$$\frac{MOD}{LOC} \geq \left(\frac{m}{\log_2 m} - 1 \right)$$

$$\text{or} \quad P \geq \left(\frac{m}{\log_2 m} - 1\right) \qquad \text{where } P = \frac{MOD}{LOC}$$

If the sort overhead is included, then

$$P \geq \left(\frac{m}{\log_2 m} - 1.4\left(\frac{m}{LOC}\right) - 1\right)$$

iv) The criterion:

Before creating an index the above condition, P, must be satisfied.

Now, from (ii) and (iii)

$$LOC \cdot m > \left[2 + \log_2 m + 1.4(1-x)m\log_2 m\right](LOC + MOD)$$

$$\text{i.e. } x > 1 - \left(\frac{1}{1+P}\right) \cdot \frac{1}{1.4\log_2 m} - \frac{1}{1.4\,m}$$

For large m

$$\therefore \quad \underline{x > 1 - \left(\frac{1}{1+P}\right)\frac{1}{1.4\log_2 m}}$$

$$\text{where } P < \frac{m}{\log_2 m}$$

It should be noted that x is a function of the usage, cost of creation and size of the index (Chapters 3 and 4),

$$\text{i.e. } x = f\left(\frac{(LOC + MOD) \cdot 1.4\, m\log_2 m}{km}\right) \text{ where k is a constant.}$$

This criterion insures that if the index is created, it will be used for locating tuples frequently enough to justify the overhead due to the sort before it is made implicit. At the same time the overall utilization of disk space is not impaired by unjustifiably keeping the indexes permanently explicit irrespective of their usage rate.

2.2 The choice of the domains to be indexed

(a) Initially we define an index for each domain of Relation R. The primary key domain (if known) is made explicit.

(b) When domain j of R is referenced the following procedure is recommended:

1) Update the reference count of index j.
   Depending on the type of reference update the locate (LOC) or the modification (MOD) count.
2) If index j is explicit go to 6.
3) Find the probability of survival x.
   Substitute x in the criterion. If the criterion is satisfied go to 5.
4) Search the domain without using an index.
5) Create index j (Index j is explicit).
6) Use the index to answer the query.
7) Stop.

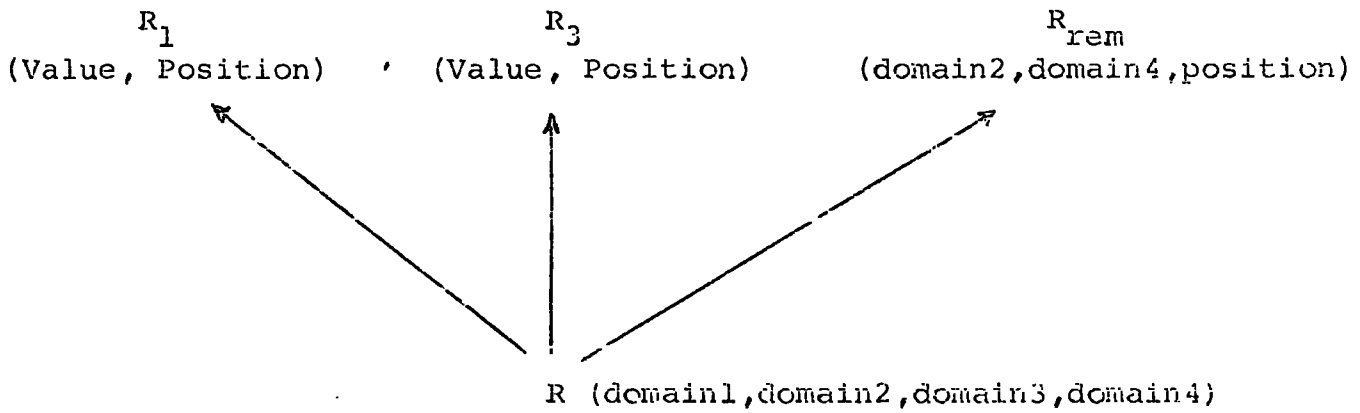
Periodically or at reorganisation time the explicit indexes are inspected to ensure that they possess the threshold ratio, P, of MOD to LOC. Accordingly some indexes may have to be forced implicit.

From the above arguments it is convincing that the choice of the domain to be indexed should be automatic. The user and the data base administrator may be able to choose correctly the primary key domain and one or two secondary key domains of a relation. Normally it is rather difficult to make any further correct choices in a multi-user system. If the automatic features of the system are carefully designed, the

process of choosing the secondary key domains must be automatic.

3. Indexing as a form of splitting

Suppose a relation R has indexes associated to most of its domains and that these indexes are formed in accordance with the above recommendations. This implies that those indexes have a high usage rate that justifies their being explicit. It is possible to add a position (or key) domain to the projection of R on the unindexed domains hence forming a relation called R_{rem}.



The indexes R_1, R_3 and R_{rem} are then made base relations and R is defined as a projection of the equijoin of all indexes and R_{rem}. This is recommended when the proposed set-up satisfies the splitting criterion discussed previously (equation (6) Chapter 5).

However, this also indicates that the formation of secondary indexes using defined relations can be a first step preceding the decision of adopting an inverted file organisation for storing a relation.

4. Summary

The above discussion shows that defined relations can be used

as secondary indexes. This solves the difficult problem of deciding which domains to be indexed and prevents the wastage of storage due to unused indexes.

CONCLUSION

The various data bases examined during this research project exhibit different characteristics, in particular their patterns of retrieval and update vary considerably. It is therefore necessary for general data base systems to be versatile enough to cope with such extremes, otherwise the data base design may be established on unfounded assumptions. Therefore automatic features should be built into data base systems so that they adapt their internal structure to the needs of the user environment.

The facility of defined relations is a powerful tool in expressing the user's view and in reflecting the effect of variations of the usage pattern on the storage. These advantages justify their inclusion in any relational data base.

The LECS criterion, which takes into account the size, cost, and frequency of reference and the dependencies among relations, is recommended as an algorithm for managing the workspace of data bases. Experiments have shown that it is reliable and theoretical analysis shows that it is a good approximation of the algorithm which gives optimum performance. The LECS has a negligible overhead compared with the order of cost incurred in data base operations.

The examination of automatic splitting has indicated that its overhead is not as costly as it seems. This automatic process tunes the storage to the usage pattern which results in significant improvement in performance.

The defined indexes offer a method for improving the response time without impairing the utilization of disk space. Indexes are created when the usage patterns, for the whole system, indicate an overall benefit to the system. This concept may be carried over to other data bases with minor modifications.

The above-mentioned gains in performance are brought about by matching the storage requirements to the usage patterns. In this manner the data base system is in a continuous process of optimizing its performance.

However, it should be mentioned that the set theoretical approach of the relational model makes the formulation and analysis of data base problems easier. Apart from its other advantages, the relational model is at least a good vehicle for research in the field of data bases.

Suggestions for further work

The following are suggestions for further work in this research area:

(i) To test the validity of defined relations in a realistic data base situation where the workspace is managed by the LECS algorithm.

(ii) A special mathematical formulation is required for the optimum choice of portions of split relations.

(iii) To test the concept of defined indexes in a real data base situation.

Finally, it is hoped that the ideas presented in this thesis are sufficiently viable to stimulate further investigation of the above concepts.

REFERENCES

ALDRED B.K.
GULLAND P.
McARTHY P.
SMEDLEY B.S.
"Interim report on the UKSC-GLC joint project", IBM(UK) Scientific Centre, to be published

BACHMAN C.W.
"The programmer as a navigator", Communications of ACM, November 1973

BACHMAN C.W.
"Data space mapped into three dimensions", Technical Manager, Data Bank Systems, Honeywell Information Systems, 1973

BELADY L.A.
"A study of replacement algorithms for a virtual-storage computer", IBM Systems Journal 5, No.2, 78-101, 1966

BELADY L.A.
PALERMO F.P.
"On-line measurement of paging behaviour by the multivalued MIN algorithm", IBM Journal of Research and Development 18, No.1, 2-19, 1974

BOYCE A.J.
"Mapping diversity: A comparative study of some numerical methods", in "Numerical Taxonomy", Proceedings of the Colloquium in Numerical Taxonomy, University of St. Andrews, September 1968, edited by A.J. Cole, Academic Press, London, 1969

BOYCE Raymond F.
CHAMBERLIN Donald D.
"Using a STRUCTURED English query language as a data definition facility", IBM San Jose (California), December 10 1973, RJ-1318

BOYCE R.F.
CHAMBERLIN D.D.
KING W.F.
HAMMER M.M.
"Specifying queries as relational expressions: SQUARE", IBM San Jose, October 16 1973, RJ-1291

BRACCHI G.
FEDELI A.
PAOLINI P.
"On relational models and languages", IFIP TC2 Working Conference on "Data Base Management Systems", Cargese, Corsica, April 1974. (Preprint)

BURNS D.
"ROBOT - A new approach to database management", Fourth European Conference on EDP Developments in Department Stores, London, November 1972

BURROUGHS
"A narrative description of the Burroughs B5500 Disk File Master Control Program", Burroughs Corporation, Detroit, Michigan, 1966

CASEY R.G.
"Design of tree structures for efficient querying", IBM San Jose, October 1972, RJ-1115

CASEY R.G.
NAGY G.
"An autonomous reading machine", IEEE Transactions on Computers, vol.C-17, No.5, May 1968, pp.492-503

CASEY R.G.
OSMAN I.M.
"Generalised page replacement algorithms in a relational data base", Proceedings 1974 ACM SIGFIDET Workshop on Data Description, Access and Control, to be available from ACM HQ, 1974

CASEY R.G.
OSMAN I.M.
"Replacement algorithms for storage management in relational data bases", to be published

CHILDS D.C.
"Feasibility of a set theoretic data structure", Proceedings of the IFIP Congress 1968 1, 420-430, North Holland Publishing Company, Amsterdam, Netherlands, 1968

CODASYL
CODASYL Data Base Task Group, Report on the CODASYL Programming Language Committee, Report CR 11, 5(70)19,080; ACM, New York, New York, 1969

CODASYL
CODASYL Systems Committee Technical Report, "Feature Analysis of Generalised Data Base Management Systems", ACM, New York, May 1971

CODD E.F.
"A relational model of data for large shared data banks", Communication of ACM, Vol.13, No.6, June 1970

CODD E.F.
"Relational completeness of data base sub-languages", Computer Science Symposium 6, May 24-25 1971, edited by Randal Rustin, Prentice Hall

CODD E.F.
"Further normalization of the data base relational model", IBM Research Report RJ-909, San Jose (California), August 31 1971

CODD E.F.
"Normalized data base structure: a brief tutorial", IBM Research Report RJ-935, San Jose (California), November 3 1971

COX D.R.
LEWIS P.A.W.
"The statistical analysis of series of events", Methuen & Co. Ltd., 1966, pp.37-58

DATE C.J.
CODD E.F.
"The relational and network approaches: comparison of the application programming interfaces", Proceedings 1974 ACM-SIGFIDET workshop on Data Description, Access and Control, to be available from ACM HQ 1974

DEE E.E.
HILDER W.
KING P.
TAYLOR E.
"Cobol extensions to handle a relational data base", Report of WP5, Advanced Programming Group, The British Computer Society, October 1973

DELOBEL C.
CASEY R.G.
"Decomposition of a data base and the theory of boolean switching functions", IBM Journal of Research and Development, Vol.17, No.5, September 1973, pp.374-386

ENGLES R.W. "A tutorial on data base organisation", Report TR 00.2004, IBM, System Development Division, Poughkeepsie, New York, 1970

EVERETT Hugh III "Generalised language multiplier method for solving problems of optimum allocation of resources", Operations Research, Vol.11, 1963, pp.399-418

GABRIELLE K. WIORKOWSKI John "A cost allocation model", Datamation, August 1973, pp.60-65

GERHARD Jaeschke "Minimal storage sorting: a comparison of different algorithms", Heidelberg Scientific Centre Technical Report, IBM (Germany), January 1974

GUIDE/SHARE Joint GUIDE/SHARE, "Data base management system requirements", W.D. Stevens, Skelly Oil Co., Tulsa, Oklahoma 74102, November 1970

IBM IBM System/360 Model 44 functional characteristics, A22-6875-6

IBM "Introduction to IBM direct access storage devices and organisation methods", IBM Trade World Corporation 1971, GC-20-1649-5.

KNUTH "The art of computer programming", Vol.3, "Sorting and searching", Addison-Wesley, 1973, pp.506-542

MATTSON R.L. GECSEI J. SLUTZ D.R. TRAIGER I.L. "Evaluation techniques for storage hierarchies", IBM Systems Journal 9, No.2, 78-117, 1970

McGEE W.C. ACM Computer Reviews, March 1974. Review no. 26,533.

McKEAY R.M. HOARE C.A.R. "A survey of store management techniques" in "Operating Systems Techniques", APIC studies in data processing, No.9, C.A.R. Hoare and R.H. Perrott Eds., Academic Press, London and New York, 1972

MTS "MTS Users Manual", University of Newcastle upon Tyne, March 1971, p.18

NOTLEY M.G. "The Peterlee IS/1 System", IBM(UK) Scientific Centre Report, March 1972, UKSC 0018

OLLE T.W. "Data structuring facilities in commercially available DBMS", Computer Bulletin, Series 2, No.1, September 1974

PALERMO F.P. "On conservatively composable relations", IBM Research Report, August 27 1970, RJ-790

PALERMO F.P. "A database search problem", IBM San Jose (California), July 1972, RJ-1072

PHILLIPS "An application example of the CODASYL-DBTG proposal", Phillips-Electrologica BV, Main Marketing Group Computer Systems, Apeldoorn, The Netherlands, June 1973

PL/1 "PL/1(F) Language Reference Manual", IBM GC28-8201-4, IBM Reference Library, 1970

RICH, Robert P. "Internal Sorting Methods", Prentice Hall, Inc., 1972

RISSANEN J.
DELOBEL C. "Decomposition of files, a basis for data storage and retrieval", IBM Research Report RJ-1220, May 1973

SENKO M.E. "Details of a scientific approach to information systems", Computer Science Symposium 6, May 24-25 1971, edited by Randal Rustin, Prentice Hall

SENKO M.E.
ALTMAN E.B.
ASTRAHAN M.M.
FEHDER P.L. "Data structure and accessing in data base systems", IBM Systems Journal, Vol.12, No.1, 1973

SIBLEY E.H. "The CODASYL data base approach: A Cobol example of design and use of a personnel file", Systems and Software Division, Institute of Computer Sciences and Technology, National Bureau of Standards, USA, February 1974

STOCKER P.M.
DEARNLEY P.A. "Self organising Data Management Systems", The Computer Journal, Vol.16, No.2, pp.100-105

SUMMERS R.C.
COLEMAN C.D.
FERNADEZ E.B. "A programming language approach to secure data base access", IBM Los Angeles Scientific Centre Technical Report G320-266, 2 May 1974

TODD S.J.P. "Implementation of the join operator in relational data bases", IEE Colloquium on "Information Structure and Store Organisation", Savoy Place, London, March 1974 (to be published)

WISHART David "CLUSTAN USER", Computing Laboratory, University of St. Andrews, St. Andrews, Fife, Scotland, 1969 (obtainable from the author)

DEFINITIONS

A: THE RELATIONAL MODEL

1. Basic definitions

The basic formal definitions are given first in section 1.1. They are followed by more informal definitions in section 1.2.

1.1 Formal basic definitions [From Codd August 1971]

(i) Given sets $D_1, D_2, \ldots, D_n$ (not necessarily distinct), R is a relation on these n sets if it is a set of elements of the form $(d_1, d_2, \ldots, d_n)$ where $d_j \in D_j$ for each $j=1,2,\ldots,n$, i.e. R is a subset of the Cartesian product $D_1 \times D_2 \times \ldots \times D_n$. D_j is referred to as the j^{th} domain of R. The elements of a relation of degree n are called tuples.

(ii) A data base is a finite collection of time varying relations of assorted degrees.

Each distinct use of a data base domain in defining relation R is called an attribute of R. For relation R the attribute names are the domain names.

(iii) Projection: The projection of R on the attribute list A designated Π_A is defined as

$$\Pi_A(R) = \{r.A : r \in R\}$$

(iv) Join: Let θ denote any of the relations $=, +, <, \leq, >$ and $\geq$. The join of relation R on domain B with relation S on domain C is defined by

$$R[B\,\theta\,C]S = \{(r \frown s) : r \in R \wedge s \in S \wedge (r[B]\,\theta\,S[C])\},$$

provided that every element of R[B] is θ-comparable with every element of S[B].

[x is θ-comparable with y if x θ y is either true or false but not undefined.]

(v) Union (∪), intersection (∩) and difference (-) are defined in the usual way. They are applicable only to pairs of union-compatible relations.

1.2 Informal basic definitions

The relation is a table of data, e.g.

PAYROLL

NAME	AGE	SALARY
SMITH	25	2000
TAYLOR	30	1500
WILLIAMS	20	1200
BEGGS	37	5000

(i) In this model the smallest unit of information is the object. Objects may be represented in the computer by integers, character strings or real numbers, etc. (e.g. SMITH, 25 and 2000 are objects). These objects are grouped together into any semantically meaningful fashion forming a set (e.g. the set of NAMEs, the set of AGEs, etc.). Such sets are termed domains (columns).

A tuple (a row) is an ordered set with one object from each domain such that a relationship exists between the objects (e.g. <SMITH,25,2000> is a tuple).

A relation is a set of all tuples of a given relationship (e.g. the above data table PAYROLL is a relation).

The degree of a relation is the number of its domains (e.g. the relation PAYROLL has a degree=3).

The cardinality of a relation is the number of its tuples (e.g. the relation PAYROLL has a cardinality=4). A Null relation has a cardinality of zero.

(ii) Relational operators:

Union: The union of two relations is the set of tuples common to both relations.

Intersection: The intersection of two relations results in the set of tuples of the two relations.

Difference: is the complement of intersection in the first relation.

Projection: generates a new relation from a given relation by subsetting and ordering the domains.

Selection: picks a subset of the tuples of a given relation satisfying a boolean expression to form a new relation of the same or lower cardinality.

Join: The join of two relations is the concatenation of a set of tuples from each relation which satisfy the boolean filter of the join.

2. The primary key

Each candidate key K of relation R is, as defined by [Codd November 1971], a combination of one or more attributes of R such that in each tuple of R the value K uniquely identifies the tuple (unique identification) and that if an attribute of K is discarded it will no longer uniquely identify a tuple (i.e. non-redundancy).

For each relation R in a data base, one of its candidate keys is arbitrarily designated as the primary key of R. Usually no tuple is allowed to have an undefined value for any of the primary key components.

3. Transitive dependence

For the relation R whose domains are A,B and C. If all three of the following time-independent conditions hold:

$$R.A \rightarrow R.B,\ R.B \not\rightarrow R.A$$

$$R.B \rightarrow R.C$$

then domain C is transitively dependent on A under R.

4. Normalisation of relations

Normalisation is a step-by-step reversible process of replacing a given collection of relations by successive collections in which the relations have a progressively simpler and more regular structure [Codd November 1971].

A relation is normalised by eliminating the following:

(1) all the domains that have tuples(sets) as elements.

(2) the non-full dependence of the non-prime attributes on candidate keys, i.e. repeating groups.

(3) transitive dependence of non-prime attributes on candidate keys.

When (1) is eliminated the relation is in the first normal form.

When (1) and (2) are eliminated the relation is in the second normal form.

When (1),(2) and (3) are eliminated the relation is in the third normal form.

An unnormalised relation is one which is not in the first mormal form. [Codd August 1971]

5. Data independence

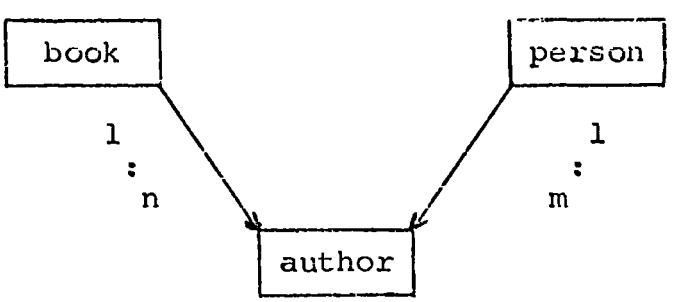
This is the independence of application programs and terminal activities from growth in data types and changes in data representation [Codd June 1970].

Data independence separates the application programs and the user's view of the data from some aspects of the storage and structure of data in the data base. Hence the application programs are protected from the changes in the external world and within the computer.

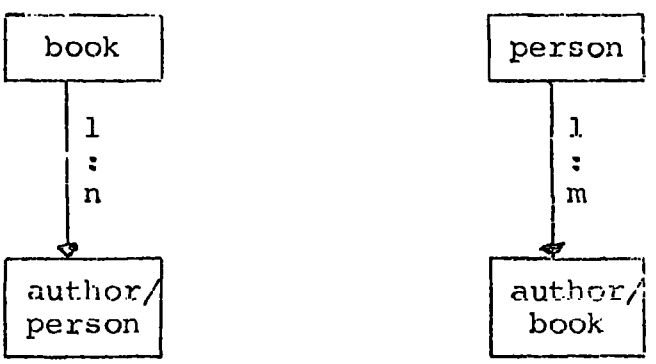
B: THE NETWORK APPROACH From[Bachman 1973]

A typical example of network structures is given below.

If the relationship between authors, books and people is considered, it may be shown that thee is an m:n relationship between books and people. Any particular book has a 1:n relationship with the persons who were its authors. From the other points of view, there is a 1:m relationship between a person and the books he may have authored. The figure shows this network relationship. It is termed a network because many people are related directly through one book, or indirectly through books with a common co-author.



This type of network is called a compound network because two different entity classes (book, person) are "bridged" by a third entity class (author). Examples of compound network elements are: book/author/person; purchase order/line item/inventory; row/element/column; resource/workload/activity. This type of network structure cannot be handled easily in hierarchical files because the dependent entity cannot be associated with more than one independent entity. The most common means of handling the above illustrated network in hierarchical structures is through duplicate files, where the network is broken into two simple hierarchies. The following diagram illustrates such a splitting of the structure of the above diagram.



If interest leads from book to author, a re-entry through the person file can indicate other books by the same author.

Network structures, then, are characterised by the fact that two entities are related through one entity of another class.

THE IS/1.0 SYSTEM

The IS/1.0 is an early version of a prototype general purpose information system. It is a single user system based on the relational model of data.

1. The instruction set of IS/1.0

The set of IS/1.0 functions and language is stack based. It handles the information in the form of relations. The stack provides the temporary storage and the workspace. Relations are conceptually loaded from the data base to the stack and operations are carried out on them independently of the information in the data base.

1.1 The stack positions

All stack positions are relative to the top of the stack. The top two relations on the stack may be referred to as:

TOP - the top item on the current stack

PEN - the penultimate item

1.2 The system operators

LOAD(R);	Loads relation R on the top of the stack.
STORE(R);	Stores the relation on the top of the stack under the name R.
LIST;	Lists the relation on the top of the stack to the output.
DROP(n);	Drops n relations from the top of the stack.

1.3 Modification of the data base index functions

NEW(R);	Introduces a new relation into the index.
DELETE(R);	Deletes a relation name from the data base index. If no synonym exists the actual data will be deleted from the data base.

1.4 The relational operators

UNITE;	Takes the top two elements of the stack, forms their union and puts the resulting element back onto the stack.
INTERSECT/DIFFER;	Forms the intersection/difference of the two elements on the top of the stack.
REORDER(domain, domain)	Is same as Projection(domain,domain).

[A logical filter is essentially a logical statement about the top two elements of the stack.

logical filter = { comparison | (logical_filter) | ¬ logical filter | logical filter and logical filter | logical filter/logical filter }]

SELECT(logical_filter) REMOVE(logical_filter)	Selects/removes from the relation on the top of the stack all tuples for which the logical filter is true.
JOIN(logical_filter)	Joins the tuples of the top two elements of the stack for which the logical filter is true.

1.5 Other operators

MIN(domain) }
MAX(domain) } Leaves a single tuple (of the relation on the top of the stack) in which the object of the parameter domain is minimum/maximum.

1.6 The IS/1.0 program

The syntax of the IS/1.0 program is:

```
BEGIN_IS/1;
  <statements of the program>
END;
```

The instructions are compiled and executed when the command

```
EXECUTE;
```

is issued.

1.7 Control commands

```
GOTO label;                    Non-conditional jump
<          >
label:
IF(logical_filter)label;       Conditional jump
```

1.8 IS/1.0 session

An example of an IS/1.0 session is shown in Figure 2.2.

2. Application programs

These are ordinary PL/1 routines interfaced with the system via the system macros and they can access the stack as well as the data base relations.

3. The defined relations capability

IS/1.0 has the facility of defining relations on other relations. An implicit relation is made explicit if the relation itself or a relation dependent on it is required for retrieval or update (cf. Chapter 2). The replacement algorithm of the IS/1.0 bases its criterion on the usage count of the relation (Cf. Chapters 3 and 4).

4.

The system may be used from a terminal or from batch (pseudo-terminal). It has backup facilities.

[Notley 1972]

Given a domain in a relation, it is required to find the set of the different object values in the domain. This may be found by either sorting the objects of the domain or without sorting. It is required to compare the number of object comparisons involved with and without sorting.

Consider a domain in a relation of cardinality m. Let the number of different objects be n.

R S

(a) Without sorting

The steps followed are:

i) take the first object from the domain and place it in the resulting domain (R).
Set i=1.

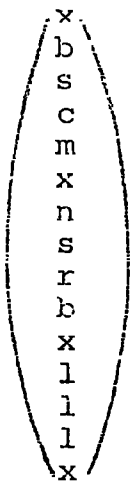
ii) take the next object from the domain. Compare it with all the objects in (R). If a match is found go to (ii) otherwise place the new object in R (and hence increasing the number of objects in R by 1, i.e. i=i+1). This is continued until all the objects of S are exhausted.

iii) i is the number of objects in R.

The average number of objects having the same value = $\frac{m}{n}$

Let us consider the number of comparisons for objects having the first value (e.g. x):

(The first object is placed without making a comparison. The following objects will be compared with the only one object placed in R)



$$= 0 + \left(\frac{m}{n} - 1\right) \times 1$$

Number of comparisons for objects having the second value (e.g. b)

$$= 1 + \left(\frac{m}{n} - 1\right) \times 2$$

Number of comparisons for objects having the third value (e.g. s)

$$= 2 + \left(\frac{m}{n} - 1\right) \times 3$$

$$\vdots$$

Similarly, number of comparisons for objects having the n^{th} value

$$= (n-1) + \left(\frac{m}{n} - 1\right) \times n$$

Total number of comparisons

$$= (0+1+2+\ldots+(n-1)) + \left(\frac{m}{n} - 1\right)(1+2+3+\ldots+n)$$

$$= (n-1) \times \frac{n}{2} + \left(\frac{m}{n} - 1\right)(n+1) \times \frac{n}{2}$$

$$= \frac{n}{2}(n-1-(n+1)) + \frac{m}{n} \times (n+1) \times \frac{n}{2}$$

$$= -n + \frac{m}{2}(n+1) \qquad = \underline{\frac{m}{2}(n+1) - n} \qquad \text{—(1)}$$

N.B. If the objects placed in R are arranged in such a way that they are ordered, this would decrease the number of comparisons to $\frac{m}{n} \sum_{1}^{n-1} \log i$. The number of replacements will increase.

(b) With sorting

The following steps are follows:

i) sort the objects of the domain.

ii) place the first object in R.

iii) compare the last object placed in R with the next object in the domain. If there is a match go to (iii). If there is no match place the compared object in R. This is continued until the objects of S are exhausted.

iv) count the number of objects in R.

Expected number of comparisons using quick sort

$= 1.3863\ m \log_2 m$ [Gerhard 1974]

Number of comparisons for scanning through the sorted domain

$= (m-1)$

Total number of comparisons

$= \underline{(m-1) + 1.386\ m \log_2 m}$ —(2)

It pays off to sort if (1) > (2)

$$\frac{m}{2}(n+1) - n > (m-1) + 1.386\ m \log_2 m$$

dividing by m (m>0)

$$\frac{1}{2}(n+1) - \frac{n}{m} > (1 - \frac{1}{m}) + 1.386 \log_2 m$$

$\frac{1}{m} << 1$

$$n+1 > 2 + \frac{2n}{m} + 2.2772 \log_2 m$$

$$n > \frac{1 - \frac{2}{m} + 2.772 \log_2 m}{1-\frac{2}{m}}$$

$$\underline{n > 2.772 \log_2 m}$$

After Casey [Casey and Osman April 1974]

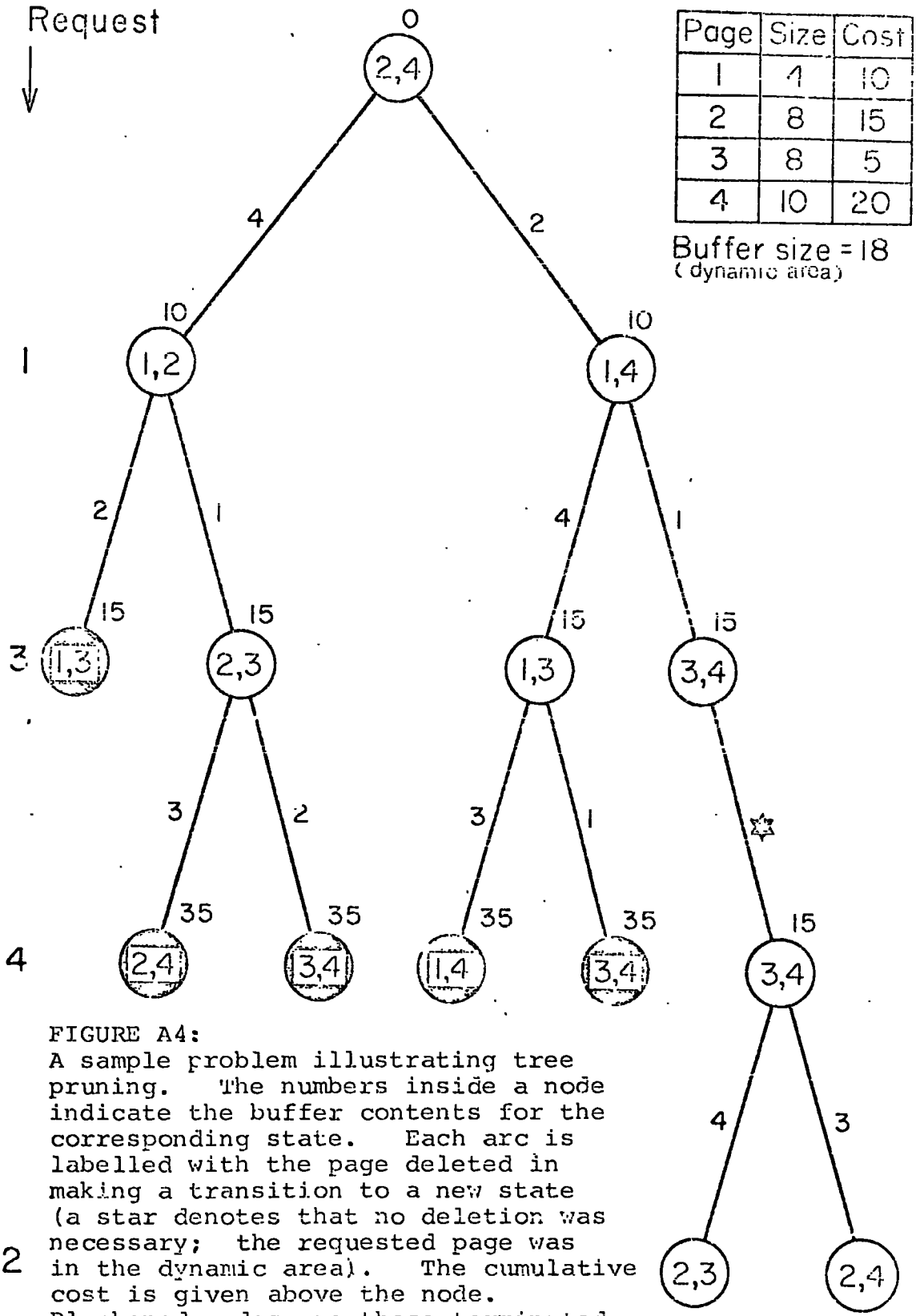


FIGURE A4:
A sample problem illustrating tree pruning. The numbers inside a node indicate the buffer contents for the corresponding state. Each arc is labelled with the page deleted in making a transition to a new state (a star denotes that no deletion was necessary; the requested page was in the dynamic area). The cumulative cost is given above the node. Blackened nodes are those terminated by the pruning algorithm.

THE PREFERRED SET

After Casey [Casey and Osman, to be published]

Assume that the frequency of request for the k^{th} defined relation over a given interval of time is u_k. Suppose also that the contents of the dynamic storage area is held fixed over this time interval, i.e. no replacement of data is carried out. This constitutes a violation of our assumed principle of database operation: however, the assumption is made only for the sake of argument. (As an aside, we observe that such an assumption is valid if requests are fulfilled by entering the desired data in a user workspace rather than a system area.)

We now seek to determine what should be the contents of the dynamic storage area in order to minimize the cost of fulfilling requests over the given time interval. We call this set of explicit relations the "preferred set". The question will be formulated as the problem of maximizing the decrease in processing cost compared with a system that answers every request from base data (i.e. does not maintain explicit forms). In mathematical notation we seek to assign values 0 or 1 to each X_k such that the total cost reduction

$$G(X) = \sum_k u_k \cdot C_k \cdot X_k$$

is maximized subject to the storage constraint

$$\sum_k S_k \cdot X_k \leq S_o$$

where S_o is the size of the dynamic area. The selection variables X_o determine the contents of the dynamic area.

For certain values of S_o the solution of this integer programming problem may be obtained using a discrete Lagrangian technique [Everett 1963]. For other values of S_o the Lagrangian method does not guarantee an optimal set of X_k, but does provide a useful bound on the cost reduction achievable.

The solution set is obtained as follows: for each k form the ratio

$$Z_k = \frac{u_k \cdot C_k}{S_k}$$

Now arrange the Z_k in decreasing order. For convenience we may assume that defined relations are actually numbered in decreasing order of Z_k. Then for each r the assignment

$$X_k = \begin{cases} 1 & k = r \\ 0 & k > r \end{cases}$$

is a solution to the maximization problem for the case

$$S_o = \sum_{k=1}^{r} S_k$$

Thus if the data base contains n defined relations this method generates solutions for n different values of S_o. Furthermore, for intermediate values of S_o, say

$$\sum_{k=1}^{r} S_k < S_o < \sum_{k=1}^{r+1} S_k$$

the obtainable savings in cost, G(X), satisfies the bound

$$\sum_{k=1}^{r} U_k \cdot C_k \leq G(X) \leq \sum_{k=1}^{r} U_k \cdot C_k + \left(\frac{S_o - \sum_{k=1}^{r} S_k}{S_{r+1}} \right) U_{r+1} \cdot C_{r+1}$$

That is, the cost savings rises no faster than linearly with storage size between the derived solution points.

The interpretation of this result is that if the time varying nature of requests is negligible (for example if requests occur randomly) then the optimal system tends to maintain in explicit form those relations having the higher values of Z_k, which is precisely the effect of the LECS technique.

Given a two-dimensional m x n matrix, F, holding the number of references of query types versus domains, it is required to cluster the groups at different levels of similarity.

In the terminology of numerical taxonomy the domains are called the forms and the query types are called the characters and the problem is a cluster analysis problem [Boyce, 1968].

Calculation of similarity

The forms are thought of as points lying in a multidimensional space, the axes of which correspond to the characters on which comparisons are based. The relative positions of the points in this "character space" are determined by the particular character values possessed by each form.

Distance coefficients

The distance coefficients are related to a class of distance functions whose general formula is (a Hölden norm):

$$d_r(j,k) = \left(\sum_{i=1}^{m} \left| F_{ij} - F_{ik} \right|^r \right)^{\frac{1}{r}}$$

where F_{ij} is the value of character i for form j, i.e. the number of references of type i query to domain j.

(If the number of characters varies from comparison to comparison the distance is usually multiplied by $n^{\frac{-1}{r}}$

The ordinary Euclidean distance (r=2) is thus

$$d_2(j,k) = \left(\sum_{i=1}^{m} \left| F_{ij} - F_{ik} \right|^2 \right)^{\frac{1}{2}}$$

The distances are evaluated. These represent the similarities between the forms. An n x n similarity matrix is constructed.

Formation of groups

We start by having n groups (subsets) each containing only one form (domain). These groups are called clusters. Each two clusters which are most similar are joined (or merged) together to form a larger cluster.

The similarity between two groups is measured by the distance between the centroids of the two groups. This distance is expressed in terms of the distances among the members of the two groups (x and y) as follows:

$$d_c = \bar{B} - \frac{t_x - 1}{2t_x} \bar{W}_x - \frac{t_y - 1}{2t_y} \bar{W}_y$$

where t_k is the number of the members of group k.

$\bar{B}$ is the mean of the squared distances between the t_k members of group k and the t_l members of group l.

$\bar{W}_k$ is the mean of the $\frac{1}{2}t_k(t_k-1)$ squared distances within group k.

At the end of each step we obtain a set of subsets. Finally, we end up with one set. The maximum number of these sets (the possible solutions) is n.