AN ABSTRACT OF THE THESIS OF

Taflan İ. Gündem for the degree of Doctor of Philosophy

in Computer Science presented on May 17, 1985

Title : View Based, Global Design of Access Structures for Relational Databases

Abstract approved: Redacted for Privacy

Earl F. Ecklund, Jr.

In this dissertation we consider the problem of automating the design of access structures for relational database systems. The main considerations are effective and rigorous utilization of the users' usage patterns, global treatment of the whole design and utilizing most of the commonly known access structures.

We represent the usage patterns on the access structures as relational algebra views. We transform a view into one or more simple access structures. Using the simple access structures we generate compound access structures. From this set of access structures we choose an approximately optimal set of access structures that process the input views as efficiently as possible and obey the storage space constraint.

In developing the transformation methods, we also develop the concepts of aggregation and generalization for access structures, and use them in obtaining compound access structures from simple data items and in integrating several access structures into one access structure. We introduced a general method that shows how any complex access structure is formed from the simple data items using aggregation.

In the transformations we consider insertion, deletion, look-up, update, building and storage space costs of access structures and/or views. The access structure types employed are tree, chain, circular chain, pointer array, cluster of tuples and sorting tuples in one or two relations. The access structures associated with a relation may cover different numbers of tuples.

For the optimization we separate all access structures into pointer and placement types. We give a 0-1 knapsack problem based approximate optimization algorithm to solve the pointer access structure optimization problem. We formulate the placement access structure optimization as a 0-1 integer programming problem. In the global optimization algorithm, we utilize the placement and pointer optimization algorithms and approximately compensate for the interdependencies between the placement and pointer access structure optimizations.

View Based, Global Design of Access Structures for Relational Databases

by

Taflan İmre Gündem

A THESIS

submitted to

Oregon State University

in partial fulfillment of

the requirements for the

degree of

Doctor of Philosophy

Completed May 17, 1985

Commencement June 1986

APPROVED:

Redacted for Privacy

Assistant Professor of Computer Science in charge of major

Redacted for Privacy

Head of department of Computer Science

Redacted for Privacy

Dean of Graduate School

Date thesis is presented May 17, 1985

Typed by Taflan İ. Gündem for Taflan İ. Gündem

ACKNOWLEDGEMENTS

I wish to express my appreciation to my thesis advisor, Dr. Earl F. Ecklund, Jr. especially for his friendship, for the interesting discussions and for creating just the right relationship that has kept me motivated and creative throughout the research and preparation of this dissertation.

I am very grateful to all of my teachers throughout my whole education and especially to my professors in O.S.U. through whom I have satisfied most of my curiosity in the area of computer science.

I thank The Scientific and Technological Research Council of Turkey for financially supporting my education in O.S.U. through a fellowship.

I feel obliged to express my gratitude to my parents for their love and support and to my friends Manuk Çolakyan and Zeki Kaya for their sincere and continual encouragement.

TABLE OF CONTENTS

I. INTRODUCTION 1

Related Past Research 1

The Main Theme and Goals 4

The Contributions 7

Overview of the Dissertation 10

II. FUNDAMENTALS OF METAMORPHOSIS 12

Views 12

Access Structures 14

Relationships 19

Explanation of the Integration Rules:

Aggregation and Generalization 20

Examples of Aggregation 24

Using Aggregation and Generalization in

Compound Access Structure Enumeration 30

Access Structure Model 36

Metamorphosis 42

III. METAMORPHOSIS OF VIEWS INTO ACCESS STRUCTURES

IN RELATIONAL DATABASE SYTEMS 44

Introduction 44

Domain 45

Range 47

First Set of Transformations 50

Costs 54

Second set of Transformations 58

IV. GLOBAL OPTIMIZATION OF THE ACCESS STRUCTURES 67

V. PLACEMENT ACCESS STRUCTURE OPTIMIZATION 72

Presentation of the Problem 72

A Solution to Placement Access Structure Optimization 78

Discussion 80

VI. POINTER ACCESS STRUCTURE OPTIMIZATION 82

Problem Statement 82

Pointer Access Structure Optimization Problem 85

A Solution to Pointer Access Structure Optimization Problem 86

VII. OVERALL ACCESS STRUCTURE OPTIMIZATION PROBLEM 91

Introduction 91

The Overall Access Structure Optimization Algorithm 95

The Rationale for the Overall Access Structure Optimization Algorithm 98

VIII. DISCUSSION 103

BIBLIOGRAPHY 107

APPENDIX 111

TWo Samples of Access Structure Generation and Optimization 111

A Very Simple Sample 112

A Less Simple Sample of Metamorphosis and the Optimization of Access Structures 130

LIST OF FIGURES

Figure		Page
1.	Coarse tree and chain aggregating compound access structure of cluster or array simple access structures	26
2.	a) A simple Network data model with record types T_0, T_1 and T_2 b) Multilist representation of the model	29
3.	The range of the resultant compound access structure in the generalization of its constituent access structures	32
4.	A complex access structure	39
5.	The ranges of the access structures depicted in figure 4	41
6.	Density functions with different costs and storage values	65
7.	The solution search space for n views	69
8.	Placement access structure modelling of a relational database	76
9.	Placement access structure model of the	

relational database in example 5.1 79

10. Dependencies of placement and pointer access structure optimization algorithms 94

11. Placement access structures with different ranges incident at a node 100

LIST OF TABLES

Table		Page
1.	Basic access structure types	16
2.	The default and representative access structures according to view types	92

VIEW BASED, GLOBAL DESIGN OF ACCESS STRUCTURES FOR RELATIONAL DATABASES

I. INTRODUCTION

Related Past Research

Access structure design is an important part of a database design. The quality of the access structures chosen directly influences the efficiency of the database operations. The importance of access structure design shows itself more clearly in relational databases than in any other commonly known database models, for the relational database model is totally independent of the underlying access structures. Due to the independence of the logical and physical parts of the relational database systems, users' access patterns play a crucial role in access structure design. Also because of this independence, it is a major task to find a suitable set of access structures that optimize the processing of the users' views (or accesses to the database).

The global design of relational database access structures is very complex like most design processes. Thus most of the work in the past and perhaps even in current practice is ad hoc as the following sentence by

Scheuermann [19] indicates. " Typical techniques for database design rely heavily on the designers' skill and experience, which is neither efficient nor effective. " Scheuermann further states that " Rules of thumb. Although widely used in practice they are misleading in many cases. " More precise and quantitative studies on this subject focus only on portions of the whole design ignoring the interdependencies of the parts or only on databases with homogeneous records, i.e. files, omitting the difficult problem of assigning access structures among sets of files.

We may categorize the past work as secondary index selection, record structuring and designs based on special access structure types. We may cite the works by Anderson [1] and Schkolnick [20] among many more in secondary index selection. The most recent and notable work on this subject is by Ip [12]. In this paper the presence of candidate index access structures and records' placement are assumed for the optimization. In record structuring techniques [15] is a good recent survey. As for special access structure optimizations, Hatzopoulos [8] presents an optimization of multilist accesss structures using dynamic programming, in a recent work.

Only in the work by Katz [13], we see the general access structure design being considered as a global

optimization problem. Although the method presented by Katz considers the globalness of the access structure design, it has the following deficiencies: it does not use the users' views effectively, the method is not easily applicable to relational database systems, instead of access structure types very general access structure characteristics are used (for example a tree and a pointer array are not differentiated) and the optimization function that is based on maximizing the frequency of traversals is kept fairly simple ignoring some of the processing costs and intrinsic interdependencies among some access structure types.

One of the most recent works on automating data structure design in computer programs is by Rowe [18]. In this work, the data structures are specified in the input program and these data structure representations are transformed into lower level storage structures, composed of cells and pointers that are like the placement and pointer elements of our primitive access structures. In our design of access structures, we infer access structures from the views and build still more complex access structures. We are not directly concerned with lower level implementations of access structures. Also we do not try to infer the semantics of compound access structures from any program automatically. The details of com-

pound access structure semantics are specified by the human designer under the guidance of the unifying concepts that we give.

One of the earlier suggestions to use views in database design is by Furukawa [5]. Recently views are used effectively and rigorously in logical database design by Roussopoulos [16] and Yao [26], and in choosing the pointer arrays for representing commonly used queries by Roussopoulos [17]. In our design, we use only a special subset of the logical views that are used elsewhere, utilize them in a unique manner and for a different purpose.

The Main Theme and Goals

The main goal of this dissertation is to develop algorithms, concepts, abstractions and methodologies that will automate the actual design and also aid the understanding of access structures for relational database systems using the following principles.

Principles :

1. Make rigorous and maximum use of the user views in every phase of the design.
2. Consider the effect of the lower physical levels of the database system.

3. Take a quantitative approach and base the design on well defined concepts and fundamentals.

4. Orient all the optimizations and decisions toward choosing access structures that will process the input set of views as efficiently as possible.

5. Consider the global dependencies throughout the design.

1. User Views :

We believe that the views are important because they are the voice and direct involvement of the users in a database access structure design. The more the information from the views is used the more efficient the resulting access structures will be in handling the users' demands.

2. Effect of Physical Levels :

In computing the various costs of the access structures several approaches may be taken. For more realistic results, computation of the costs should be system dependent and reflect the particularities of the lower level software and hardware. Also a large variety of costs should be included so as to show the differences among access structures and reflect most of the user activities on the database.

3. Well Defined Concepts :

Well defined concepts and terms cause less ambiguity and together with the abstractions allow better comprehension. We also express various properties of the access structures and views quantitatively to enable more precise comparisons.

4. Process Efficiently :

The main objective in all the optimizations in the design is to minimize the cost of processing the input set of views. This objective is difficult to optimize and thus has been avoided in global optimization of access structures in the literature. However we believe that it is the most meaningful criterion in choosing the access structures. Scheuermann [19] has a similar opinion as can be seen from the following statement " ... a weighted function of access and storage costs would constitute a more realistic performance measure. " Some of the other performance measures would be just considering processing costs or storage space or reconfiguring costs.

The difficulty in our optimization arises from the extra interdependency between pointer and placement access structures caused by this objective function. The objective function based on the frequency of traversal of the access structures does not have this extra interdependency

and thus the resulting objective function is easier to optimize. However it is not a good design criterion due to the fact that the set of access structures that maximize the total frequency of traversals do not necessarily minimize the cost of processing the input views (that is insertion, deletion and look-up costs of views and building, update and storage costs of access structures). We would like to mention that the access structure selection algorithms developed for the dissertation would also work for maximizing the total frequency of traversals.

5. Global Dependencies :

Local optimization methods will not yield good results due to various interdependencies among access structures, constraints, and conflicting choices involved in the optimization.

The Contributions

The major contributons of this dissertation may be summarized as follows

1. Use of aggregation and generalization concepts in compound data structure and database access structure design and analysis: These concepts increase our understanding of complex data structures by dividing them into simpler

parts and relating simpler parts to their semantics. Once we know the semantics of the simple parts, we understand the semantics of the larger parts more easily. They also make possible automatic data structuring and generating all possible useful compound structures from simpler parts and thereby making the optimization of data structures possible.

2. The concept of metamorphosis of views into access structures in relational database systems: this is a systematic way of producing a good or potentially best set of access structures that represent the views. Metamorphosis gives us an answer to the question: " How do you choose, from all possible access structures, the ones that are needed given a set of user views? ". Metamorphosis does systematically and quantitatively what happens in the minds of the database designers in an ad hoc manner. As a result of metamorphosis, access structures are created only if they are needed. The set of candidate access structures change according to the set of views given and do not stay the same just because the logical database is not changed, as is the case with the known approaches. Several access structures may be used in processing all the views related to an attribute instead of just one. Metamorphosis also gives us a better understanding of the

access structure creation. It is a conceptual framework that shows a clear picture of access structure creation.

3. An integer programming formulation of the problem of choosing an optimal subset of a given set of placement type of access structures (i.e. clustering a relation, sorting a relation and linking two relations by placing together identically valued tuples on a specific attribute value) of a relational database.

4. An approximate index selection algorithm with a lower worst case time order, O(n log n) than that, O(n ** 2) published in the most recent paper on this subject, namely Ip [12].

5. The observation that pointer and placement access structures are partially interdependent in a global access structure design that is based on minimizing the cost of processing the views: we give an approximate algorithm to compensate for this interdependency.

Some of the other contributions of the dissertation are summarized in the following. The relationship between views and simple access structures and that between simple and compound access structures are illuminated. Commonly known access structures are categorized. Different types of access structures and access structures that cover all

or only some of the tuples in a relation are treated together in the same design.

In general the dissertation illuminates some of the overlooked issues in access structure design; the concepts and abstractions developed are applicable to any data model and in fact to compound data structures in general as shown by examples in chapter 2. Finally the algorithms presented in the dissertation are coded and tested.

Overview of Dissertation

In the order of appearance, the contents of the rest of the dissertation are as follows. In chapter 2, we explain the fundamental concepts, abstractions, definitions and the principles that form the basis of the metamorphosis of the views into access structures. The third chapter covers the metamorphosis as applied to relational database systems using a subset of the access structure types that we define in chapter 2. The fourth chapter introduces the problem of global access structure optimization given a set of candidate access structures and explains our approach to the problem. Chapters five, six and seven explain the algorithms used in our approach to solving the global access structure optimization problem introduced in chapter four. The fifth chapter

presents the optimization of the placement access structures. In the sixth chapter we explain the optimization of the pointer type of access structures. The seventh chapter covers the interdependency of placement and pointer access structure designs. Finally the eighth chapter covers the discussion and further research.

II. FUNDAMENTALS OF METAMORPHOSIS

In this chapter we explain the fundamental definitions, abstractions, concepts and the principles used in the access structure metamorphosis of the user views. Informally metamorphosis is the process of changing views into access structures that represent these views. In the abstraction of this process the basic entities are the views and the access structures.

Views

Let us consider the views first. As we mentioned earlier, the views we use in our design are only a subset of the views defined elsewhere. A view in general is a sequence of relational algebra operations that may be represented by a query tree (see the explanation by Ullman [24]). In our model we are interested only in those views that are performed directly on the secondary storage records. Since our main concern is on access structure design, we are concerned only with those parts of a query tree that are directly related to the secondary storage structures. However processing the views or accessing the database efficiently has an important effect on the total processing cost of a query.

We represent a view by a single relational algebra operation. The complete description of a view as we use it is given by the following definitions.

<u>Definiton 2.1</u> Let R be the set of all relations in a physical database. Let B be the set of binary relational algebra operations containing intersection, difference, union, join, quotient and cartesian product. Let b be the set containing all b_i such that $b_i \in B$ and $b_i : (R_j, R_k) \dashrightarrow S_i$, where $R_j, R_k \in R$ and S_i is the set of physical tuples generated by b_i.

Let U be the set of unary relational algebra operations containing selection and projection. Let u be the set containing all u_i such that $u_i \in U$ and $u_i : R_j \dashrightarrow T_i$, where $R_j \in R$ and T_i is the set of physical tuples generated by u_i.

Let $a = u \cup b$. A <u>view</u> is an element of a.

Views can be parameterized by the following properties:

1. the average number of physical tuples that it covers;
2. the frequency of the future use of the view;
3. the type of access (i.e. insertion, deletion and look-up) associated with the view.

Definition 2.2 The range of a view is the set of physical tuples (storage records) generated by it (that is S_i or T_i as used in Definition 2.1).

A view describes the set of tuples that are associated to each other and accessed together. It is interesting to note that the quantitative values associated with a view are based on statistical observations. The properties of a view come from the logical database schema, extension of the schema and usage information sources. Finally, we note that our use of a view in representing secondary storage accesses is novel.

Access Structures

The other important entity in our model is the access structure. We define an access structure as follows.

Definition 2.3 An access structure is a placement of data items and/or extra data structures that can be acted upon to access data items efficiently.

Definition 2.4 The range of an access structure is the set of physical tuples that can be accessed by the access structure.

An access structure increases the efficiency of one or more <u>access purposes</u> which may be update, look-up, insert or delete for a set of data items that are accessed by the access structure. We see a large variety of access structure types in the computer science literature. We divide basic access structures into three categories <u>placement</u> , <u>pointer</u> and <u>transformational</u> as described in the following.

<u>Definition 2.5</u> <u>Placement</u> type of access structures are those that provide contiguity of data objects within physical storage blocks.

<u>Definition 2.6</u> <u>Pointer</u> type of access structures are those that involve only extra data structures which point to the location of the data objects.

<u>Definition 2.7</u> <u>Transformational</u> access structures are those that involve transforming a key (related to the values of the data objects) to the location of the data object and that cause the data objects to be stored only according to their key values in physical storage.

We further categorize the basic access structures into the following more detailed categories. Table 1 shows the basic access structure types.

Table 1. Basic access structure types

Placement	Pointer	Transformational
1. cluster	1. tree	1. hashing
2. well-placed	2. pointer-array	
3. sorted	3. circular-chain	
	4. chain	
	5. multiattribute tree	

a) Placement Types

i) Cluster. The related data items are placed together without a predetermined pattern. (This definition of cluster may not correspond to its usage in the literature.)

ii) Well-placed. A data item is placed next to the data items that it is related to in a one to many relationship. This is different from cluster in the sense that the relationship among data items that are placed together is fixed (that is, one to many). This access structure is used only for linking in relational databases.

iii) Sorted. The data items are clustered and sorted according to a certain attribute.

b) Transformational

i) Hashing. A key is transformed into one of the bucket addresses where the data are stored.

c) Pointer types

i) Pointer adjacent. The pointer is adjacent to the data item. In this category we have chain and circular chain. In circular chain the last pointer points to the first item.

ii) Pointer separate. The pointer is separate from the data item. In this category we have tree, pointer array and multiattribute tree. Tree is subdivided into two: coarse tree and fine tree. In a coarse tree each pointer points to a set of data items whereas in a fine tree each pointer points to a single data item.

We leave further subclassifications to lower levels of the design. For example the category tree access structure can be classified into various different sorts such as B-tree, binary tree, AVL tree, etc.. One also notices that only one access structure type among various multiattribute access structures is chosen here. We mainly left the multiattribute access structure types outside our focus, as they are not being used commonly in general pur-

pose database systems.

In our categorization we bring together access structures with similar abstract characteristics thus decreasing the number of access structures to be considered in a design. The resulting types are just detailed enough to show the differences among themselves in processing different sets of views. In other words given a set of views there is an access structure type such that an access structure from that type processes the input set of views more efficiently than an access structure from any other type.

From the view point of mathematical relations, the categorizations of access structures may be explained as follows. The access structure types shown on table 1 can be considered equivalence classes. In partitioning the set of access structure types into these equivalence classes, we use the equivalence relation " processes the certain set of views with a similar efficiency as ". Thus access structures with similar processing cost functions are brought together.

We categorize all access structures as simple, compound or complex access structures. <u>Simple access structures</u> are those that access only data items. <u>Compound access structures</u> are those that use the simple access structures in their constitution. As for <u>complex access</u>

structures, they contain simple access structures and compound access structures in their constitution. Simple access structures are the building-blocks and also the structuring entities of compound access structures. Furthermore, we conjecture that all compound access structures are generated from simple access structures using aggregation and generalization concepts (explained on page 21f). An example of a complex access structure would be clusters of data items connected to each other by a chain and a tree pointing to each cluster as shown in figure 1 (page 26). Clusters of data items connected to each other by a chain and a tree pointing to each cluster are two separate compound access structures. We specify compound access structures more rigorously in the section on access structure modelling (see page 36).

The access structures also have a set of properties which are dependent on the application and the goal function to be optimized. In our design some of the properties are the list of views associated with an access structure and the cost of processing these views.

Relationships

Access structures and views are related to each other mainly through the tuples in their ranges. We make use of

the following relationships in the metamorphosis:

1. Every view can be represented by a set of data objects.

2. Every access structure can be represented by a set of data objects.

3. Every view can be represented by an access structure.

4. Every access structure can be represented by a set of views.

Explanation of the Integration Rules : Aggregation and Generalization

Basic rules of integration of views into access structures make use of the aggregation and generalization concepts as we explain in this section. Aggregation and generalization concepts were first introduced by Smith [21]. These concepts and their use in logical data modelling are explained by Tsichritzis [23]. Smith [22] gives an application of aggregation in designing a new abstract data type. Our use of a variation of these concepts in general access structure design is novel. Although our definition of aggregation and generalization is different,

we are somewhat influenced by the interpretation of these concepts by Tsichritzis [23].

Aggregation as we define it seems to explain all the compound access structures that we found in the literature. Aggregation and generalization seem to be generalizing concepts that unify the efforts in compound access structure design under a single framework. These concepts are not only helpful in understanding the existing compound access structures but are also useful in creating new compound access structure types from simple access structures. We use aggregation and generalization in systematically generating all possible useful compound access structures from a given set of simple access structures. We believe that they may also be used in creating new general purpose data structures from the existing ones. By making use of these concepts as we define them, one should be able to generate access structures or data structures to meet specific goals.

Generalization of access structures is forming a compound access structure by bringing together simple access structures that have the same function as the resultant compound access structure. Consequently they are not retained in the resultant compound access structure. In a way generalization is viewing a set of access structures as one under a certain set of characteristics or func-

tions. Generalization is useful in replacing a set of simple access structures with small ranges by a generalizing access structure that has a much larger range. From one view point, it is abstracting out some of the detailed functions of the constituent access structures in order to bring them together. Generalization is very useful in access structure metamorphosis of views.

Aggregation of access structures is forming a compound access structure by bringing together simple access structures with different functions which are made use of in forming the overall function of the compound access structure. Thus the constituent access structures are retained in the resultant access structure. From this view point the constituent access structures are considered as part of the whole. We can say that aggregation is constructing a compound access structure from its constituent simple access structures. The constituent simple access structures are brought together under the structure of an aggregating access structure. In other words the relationship among the constituent parts are determined by the structure of the aggregating access structure.

Both in aggregation and in generalization, the constituent access structures may be of the same type, homogeneous , or of different types, heterogeneous. Also, we can have the generalizations and aggregations based on

either the semantics or syntax of the simple access structures. In heterogeneous generalization we consider the sementics of of the constituent access structure.

In generating or analyzing compound access structures using aggregation or generalization, it is also important to know and utilize the semantics of compound access structures. In the semantics of a compound access structure, we mainly examine:

i) the goals of the compound access structure;

ii) the conditions imposed on the constituent access structures due to certain properties of the generalizing or aggregating access structures.

The goals are related to the costs of various types of access purposes. In the relational database access structures, they may be processing relational algebra operations with respect to access purposes as efficiently as possible; optimizing the secondary storage space or minimizing the number of access structures. The conditions imposed by the generalizing or aggregating access structures are related to the range of the constituent access structures.

Both aggregation and generalization are based on bringing together simple access structures in accordance with an abstraction. Using the same access structures, we

may end up with a different compound access structure depending on the semantics. In each compound access structure, we consider different properties of the constituent access structures. This is due to different goals.

The aggregation access structures mentioned here are of two levels: the aggregating level and the constituent level. It is of course possible to have more levels of aggregation where the compound access structures become constituent access structures. However the database relations have to be extremely large in order to make such access structures feasible. In our design, we did not consider them to be realistic.

Although the number of different compound access structures that can be generated from a set of simple access structures using aggregation and generalization may be very large, the number of useful (i.e. cost effective) compound access structures is relatively small. The semantics of compound access structures eliminate those access structures that do not conform to the restrictions or that do not help towards the goals.

Examples of Aggregation

In this section we are going to analyse briefly some of the compound access structures found in the literature

from the view point of aggregation and generalization.

A. The generalized access structure introduced by Haerder [7] is an example of a set of two nonconflicting aggregation compound access structures sharing the same set of constituent access structures. In a database we may see more nonconflicting compound access structures on the same set of simple access structures. The two compound access structures here are a coarse tree aggregation of a set of array or cluster simple access structures and a chain aggregation of the same set of simple access structures as shown in figure 1. In functional notation these compound access structures may be shown as

```
STRUC ( COARSE-TREE ( CLUSTER ( data items )),
        CHAIN ( CLUSTER ( data items ))
```

or

```
STRUC ( COARSE-TREE ( ARRAY ( data items )),
        CHAIN ( ARRAY ( data items )) ,
```

where STRUC stands for an unnamed aggregating access structure. As for the semantics of the compound access structure, some of the goals may be summarized as follows:

i) decreasing the access time for a selection operation involving the tuples in the range of the simple access structures;

ii) fast sequential scan of these tuples;

iii) fast linking of the tuples in two or more relations (i.e. equi-join relational algebra operation).

The other issue in the semantics is the conditions imposed on the constituent access structures because of the properties of the tree and chain aggregating access structures. The range of this access structure in a relational database is composed of the relations that share the same attribute used in the tree.

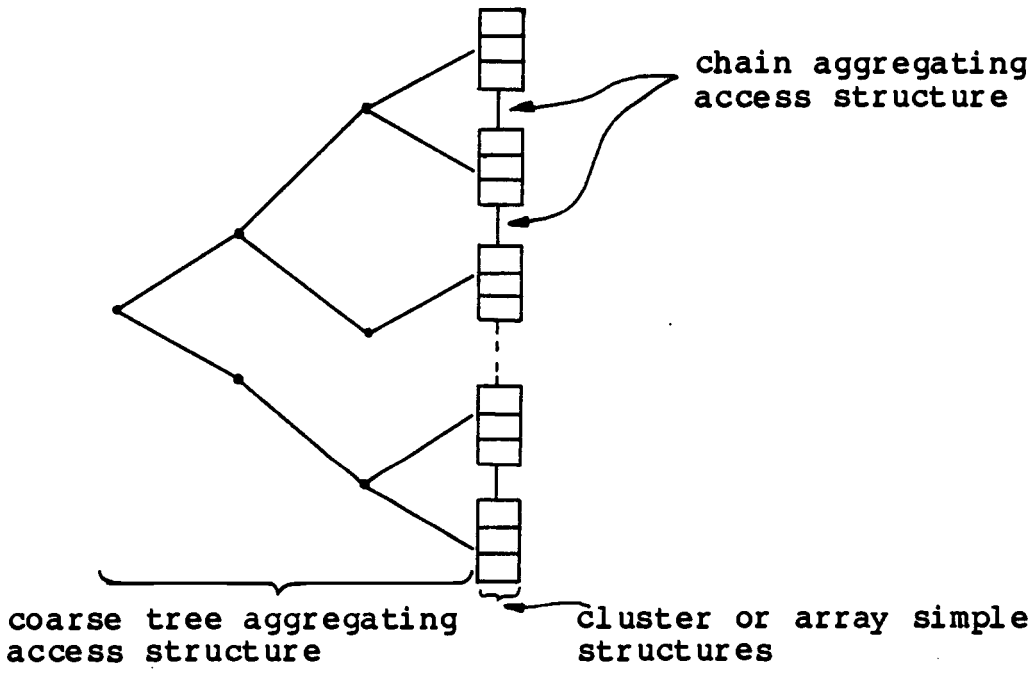


Fig. 1. Coarse tree and chain aggregating compound access structure of cluster or array simple access structures.

B. Lipton [14] shows the use of a hashing scheme for storing a broad class of data structures. The examples given by Lipton may be explained by aggregation. The constituent access structures here comprise the buckets of the hashing aggregating access structure. The constituent access structures are homogeneous and of type tree, array or a variation of an array. In functional notation these compound access structures may be represented as

HASH (TREE (data items)),

HASH (ARRAY (data items)),

HASH (ARRAY-LIKE STRUCTURE (data items)).

C. Chang [4] presents an extended k-d tree organization for performing multiple attribute clustering in databases. The compound access structure of Chang is a multiattribute tree aggregation of cluster access structures. Here again the constituent access structures are of homogeneous type. In functional notation this compound access structure may be shown as

MULTIATTRIBUTE (CLUSTER (data items)).

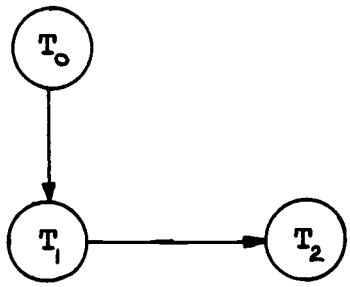
The most important characteristic in the sementics of this data structure is its having dynamic record clustering.

D. Perhaps the most commonly known example of compound access structures is found in network database access structure implementation. Such an implementation

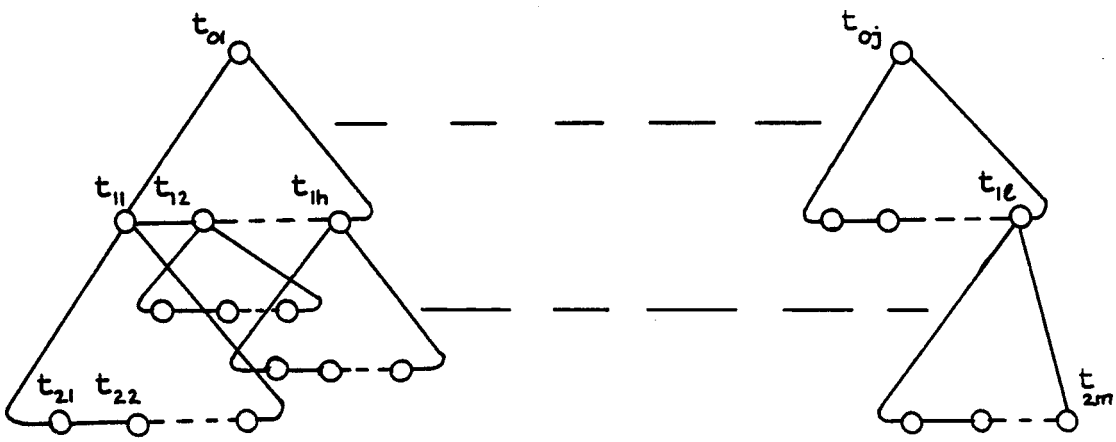
using multilist organization is a cluster aggregation of circular chain simple access structures and circular chain aggregation of circular chains compound access structures. For example the record-type relationship shown in figure 2.a can be represented by

```
CLUSTER ( CIRCULAR-CHAIN
          ( CIRCULAR-CHAIN ( data items )))
```

aggregation access structure as shown in figure 2.b. Obviously this compound access structure could be used in a hierarchical database implementation too.



a)



$t_{oi} \in T_o$, $i=1,j$
$t_{1i} \in T_1$, $i=1,l$
$t_{2i} \in T_2$, $i=1,m$

b)

Fig. 2. a) A simple Network data model with record types T_o, T_1 and T_2. b) Multilist representation of the model.

Using Aggregation and Generalization in Compound Access Structure Enumeration

We use aggregation and generalization concepts in view metamorphosis to generate all possible useful compound access structures from a set of simple access structures in relational database systems. Ultimately a subset of these access structures that optimize an objective function are chosen. Aggregation and generalization help us enumerate all possible compound access structures in a systematic and orderly fashion. More specifically, aggregation helps us in generating compound access structures that decrease the search time for a large collection of constituent simple access structures. Generalization is useful in generating compound access structures that decrease the number of simple access structures which consequently help decrease the secondary storage needs and the search time for access structures.

In metamorphosis for relational databases, we obtain simple access structures that are used for selections associated with attributes and for linkages between two attributes in two distinct relations (for example, for join operations). For selections, we consider the access structures based on each attribute separately. That is for each attribute, we use aggregation and generalization in bringing together access structures used for selecting

tuples based on their values for that specific attribute. For linkages, for each pair of attributes we use generalization in bringing together simple access structures used for join operations. In the following we consider selection and linkage access structures separately.

A. Selection

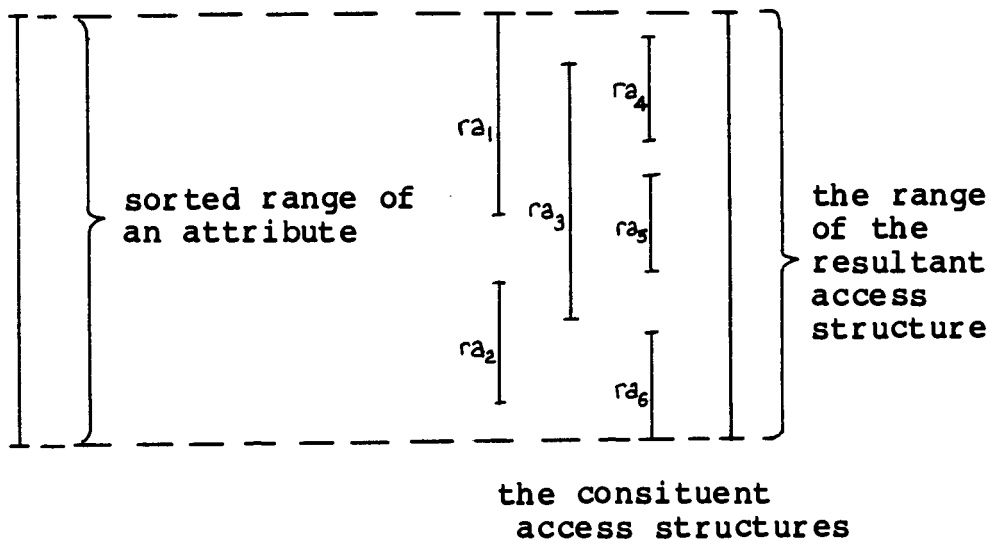
We assume that we have k simple access structure types, h of which can be used for selection.

Generalization:

In the homogeneous case, for each attribute the number of different generalization access structures is h. That is each access structure type acts as a generalizing access structure for the homogeneous constituent access structures of its own type. For example a collection of trees using the same attribute as the key are replaced by a single tree. In the heterogeneous case we have another h access structures. Here again each access structure type acts as a generalizing structure but this time, for different types of constituent access structures that are used for selections on that attribute.

In the semantics of the resultant compound access structures, the range considerations are worth looking into. Let the range of an attribute be the values taken

by the tuples of the relation on that attribute. The range of the resultant compound access structure is the union of the ranges of its constituent simple access structures. One example is shown in figure 3.



ra_i stands for the range of the constituent access structure i.

Fig. 3. The range of the resultant compound access structure in the generalization of its consituent access structures.

The goal of generalization is to decrease the number of access structures with small ranges. Consequently, this decreases the secondary storage requirements for pointer type access structures directly and for all access struc-

ture types indirectly. This indirect savings is due to decreasing the chances of conflicts by decreasing the number of access structures. A conflict arises when two placement access structures sharing tuples in their ranges want to have two different orderings of the tuples. Since conflicts are sometimes resolved by duplicating the tuples that are shared by the conflicting access structures (Katz [13]), extra storage required for conflict resolution is saved.

The number of access structures is further decreased by eliminating the compound access structures on the basis of the number of tuples in their ranges. The ones with a number of tuples under a certain threshold number are eliminated. This threshold number is determined by the efficiency considerations of the generalizing access structure type. For example it is not worth having a tree access structure to search for five data objects.

Aggregation:

In the homogeneous case, we have h*h different access structures. Each one of the h access structures acts as an aggregating access structure for a homogeneous set of each of h access structure types. The aggregating access structures are of selection type and the constituent access structures may be of any type (but they are all of selection type if they are used for selection). If the

constituent access structures are heterogeneous, there are h aggregating access structures. Each access structure type acts as an aggregating structure for the constituent access structures that are used for selections on the same attribute.

The ranges of the constituent access structures have to suit the requirements of the aggregating access structure. For cluster, circular chain and chain, there are no requirements on the ranges of the constituent simple access structures. For key (a key being an attribute value used for the search) dependent access structures (i.e. sorted, hashing, array, tree), we have to consider the linkage and selection constituent access structures separately. For linkage type access structures the value of the owner tuple is used as the key. For selection type access structures, we choose the starting values of the ranges of the constituent access structures as keys. However for selection type constituent access structures, a multiattribute access structure is a better aggregating access structure because having the starting and ending values of the ranges of the constituent access structures as the keys helps in further differentiating among the constituent access structures.

The main goal of aggregation is decreasing the search time for the constituent access structures. Because of

this, aggregation is cost effective only when the number of constituent access structures is large.

B. Linkage

Here the use of generalization is restricted. Generalization does not generate efficient access structures for linkage, unless the generalizing access structure is on the same range as its constituent simple access structures. The reason is that we need to keep the constituent access structures in the resultant access structure. If not, the cost of processing is increased considerably. In linkage we have h * (k - h) possible homogeneous aggregations and k - h generalizations.

The number of possible aggregations and generalizations are limited by the following semantic factors:

i) the number of constituent access structures;

ii) the requirements imposed on the constituent access structures by the aggregating access structures such as sortability;

iii) the ranges of the constituent access structures;

iv) the function of the constituent access structures which can be explained in terms of relational algebra operations;

v) the goals of the compound access structure;

vi) the complexity of the optimization algorithm to process these access structures.

Access Structure Model

In our access structure model, we have the following entities:

i) data items,

ii) simple access structures,

iii) compound access structures.

There is a relationship between any two of these entities. Simple access structures are formed from the data items using the previously explained simple access structure types. Compound access structures are built from simple access structures using aggregation. The range of a compound access structure consists of the data items that are in the union of the ranges of the constituent access structures.

It appears that all known access structures can be built using only aggregation.

Conjecture 2.1 :

Any complex access structure that has appeared in the

literature can be built by the aggregation of simple access structures.

In each case above the aggregating access structure is a basic access structure or an aggregating structure (STRUC).

STRUC is used for describing the associations among access structures that are not described by the basic access structure types that appear on table 1. The relationship among the constituents in such cases has to be determined from the relationships on the ranges.

More precisely, in functional notation, any access structure x can be expressed as follows:

x = f(A), where f is a basic access structure type or a STRUC and

$$A = \begin{cases} \{ d_i \mid d_i \text{ is a data item, } i \in (1,n) \} \\ \qquad\qquad \text{or} \\ \{ s_i \mid s_i \text{ is defined as } f(A), \ i \in (1,n) \} , \end{cases}$$

n being a positive integer.

We read x = f(A) as x is an f access structure formed from data items, d_i , or as an f aggregation of constituent access structures, s_i, depending on A.

The range of an access structure x is expressed by R(x). The relationship among the ranges of access structures is expressed using the set operations:

i) union of R(x1) and R(x2) (U(R(x1), R(x2)));

ii) intersection of R(x1) and R(x2)
(I(R(x1), R(x2)));

iii) difference of R(x1) and R(x2)
(D(R(x1), R(x2)));

and the inference

iv) R(x1) subset of R(x2) (S(R(x1), R(x2))).

In functional notation any set of tuples y in the range of an access structure is expressed by

$$y = \begin{cases} \{ d_i \mid d_i \text{ is a data item, } i \in (1, n) \}, \\ \quad n \text{ being a positive integer,} \\ \quad \text{or} \\ op(y_1, y_2), \text{ where op is one of the operators} \\ \quad \text{defined above and } y_1, y_2 \text{ are of the same} \\ \quad \text{form as } y. \end{cases}$$

<u>Example 2.1</u>

The access structure Xt in figure 4, can be expressed

by means of aggregations using functional notation as follows.

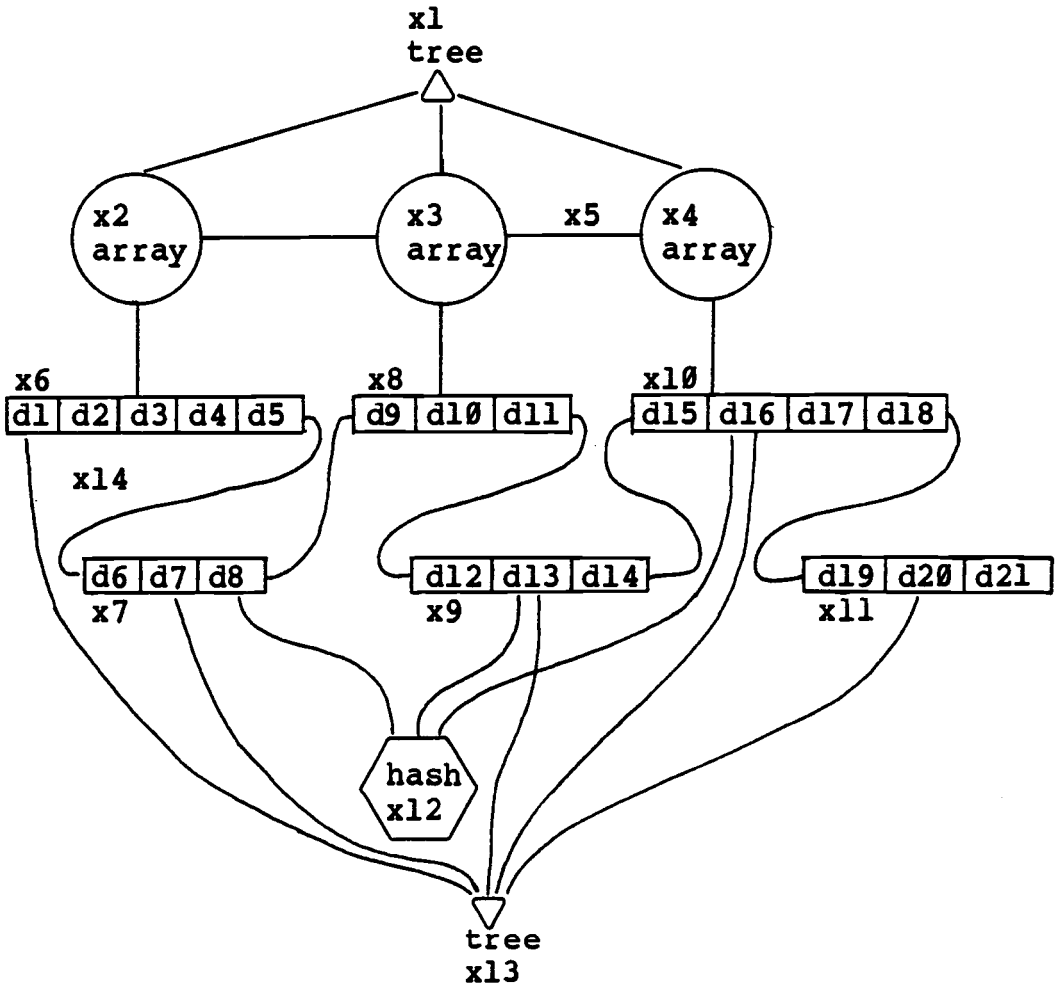


Fig. 4. A complex access structure.

x6 = SORTED (d1, d2, d3, d4, d5),

x7 = CLUSTER (d6, d7, d8),

x8 = CLUSTER (d9, d10, d11),

x9 = CLUSTER (d12, d13, d14),

x10 = CLUSTER (d15, d16, d17, d18),

x11 = CLUSTER (d19, d20, d21),

x2 = ARRAY (d1, d2, d3, d4, d5),

x3 = ARRAY (d9, d10, d11),

x4 = ARRAY (d15, d16, d17, d18),

x1 = TREE (x2, x3, x4),

x5 = CHAIN (x2, x3, x4),

x14 = CHAIN (x6, x7, x8, x9, x10, x11),

x12 = HASH (x7, x9, x10),

x13 = TREE (x6, x7, x8, x9, x10, x11),

Xt = STRUC (x1, x13, x12, x5, x14).

The ranges of the access structures that are specified in the following are depicted in figure 5.

R(x6) = { d1, d2, d3, d4, d5 }

R(x7) = { d6, d7, d8 },

R(x8) = { d9, d10, d11 },

R(x9) = { d12, d13, d14 },

R(x10) = { d15, d16, d17, d18 },

R(x11) = { d19, d20, d21 },

R(x2) = R(x6),

R(x4) = R(x10),

R(x3) = R(x8),

R(x1) = U(R(x2), R(x3), R(x4)),

R(x5) = R(x1),

R(x14) = U(R(x6), R(x7), R(x8), R(x9), R(x10), R(x11)),

R(x13) = R(x14),

R(X12) = U(R(x7), R(x9), R(x10)),

R(Xt) = R(x13).

The range specifications and the access structure specifications together completely specify the complex access structure in figure 4. We can also make the following inferences to show the use of the set operations.

1. R(x10)= I(R(x1), R(x12))
2. S(R(x12), R(x13))
3. U(R(x11), R(x7), R(x9)) = D(R(x14), R(x1))

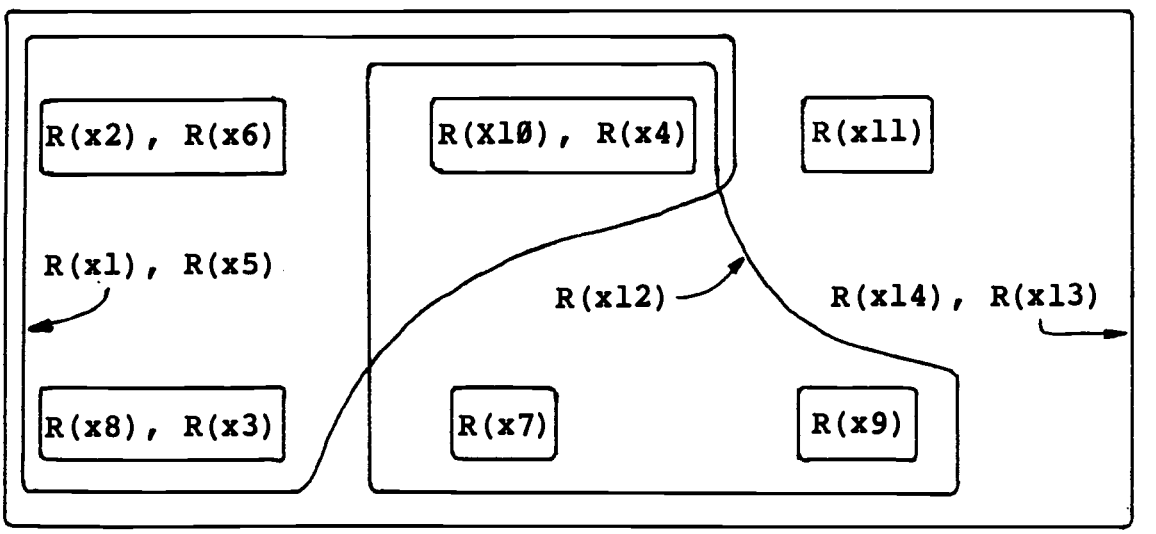


Fig. 5. The ranges of the access structures depicted in figure 4.

Metamorphosis

Metamorphosis is a conceptual framework that enables us to use the concepts and the abstractions explained in this section to systematically transform a set of views into a corresponding set of access structures.

> Definition 2.8 Metamorphosis is a sequence of transformations applied to a set of initial objects and that changes them into a set of target objects. The inverse transformations are not necessarily possible.

Metamorphosis has the following properties associated with it:

1. domain : the set of initial objects;

2. range : the set of target objects;

3. intermediate objects : objects that are formed during the metamorphosis vbut are not target objects;

4. transformations : rules for changing the initial objects to intermediate or target objects and the

intermediate objects to other intermediate or target objects.

Metamorphosis can be applied to a set of object types or a set of instances of object types. In the next chapter, we give a detailed example to the application of metamorphosis in relational database systems.

III. METAMORPHOSIS OF VIEWS INTO ACCESS STRUCTURES IN RELATIONAL DATABASE SYSTEMS

Introduction

Now that we have explained the general principles and the fundamental concepts of access structure design in general, we are going to consider access structure design only for relational databases starting with this chapter. We use metamorphosis in obtaining access structures for relational database systems. Metamorphosis and other concepts presented in the previous chapter may be used in various ways to obtain access structures for relational databases. In this example of metamorphosis, we assume that the physical database schema remains fixed. The material we present in this chapter is one way of using metamorphosis. In our choice of various options such as access structure types and their ranges, we are influenced by the degree of intractability of optimizations and the size of their search spaces.

The main goal of metamorphosis is to generate a good set of candidate access structures from a set of input views, using the local knowledge about the views. A subset of these access structures is chosen later by the optimizer considering the global knowledge. Local

knowledge alone is not sufficient to generate the globally optimal set of access structures. Thus in metamorphosis we generate a set of alternative access structures corresponding to a view or a set of views. Each of these access structures is the most efficient representation of the set of views under certain conditions.

The sections of this chapter are based on the properties of the metamorphosis. We also include a section on the costs of access structures since it is a prerequisite for the section on the second set of tranformations. The metamorphosis procedure takes a set of input views, specified in the section on the domain, and transforms them into access structures, whose types are specified in the section on the range of metamorphosis. The rules used in the transformations are explained in the transformations sections.

Domain

The domain of metamorphosis contains a set of views that reflect the anticipated future use patterns of a relational database. In chosing the types and the values of the properties of the views in the domain, the designer has a latitute to influence the outcome of the metamorphosis, i.e. the access structures.

The views in the domain are of the relational algebra operation types explained in the following (for an explanation of the semantics of these operations, see for example Ullman [25]). In the following R and S stand for relations.

1. Selection (F, R), where F is a formula of the form A.op.C where A is an attribute name in R; C is a constant and op is an arithmetic comparison operator: <, >, =, <>, <= or >=.

2. Projection (R) (Note that the attribute name involved in the definition of projection is not needed because in our design it is assumed that the tuples are not partitioned as is the case in System R [2].).

3. Cartesian product (R, S).

4. Union (R, S).

5. Set difference (R, S).

6. Intersection (R, S).

7. Join (H, R, S), where H is a formula of the form A.op.B where A and B are attribute names from relations R and S respectively and op is the same as that in selection.

8. Quotient (R, S).

Additionally each view has the properties explained in chapter 2.

Range

In the range of our metamorphosis, we have simple and compound access structures. In our choice of simple access structures, we are influenced by the choice of access structures in Relational System R [2]. The compound access structure types we have are obtained from the simple access structures using the previously introduced aggregation and generalization concepts.

In our design we use the following access structure types:

1. cluster,
2. sorted,
3. well-placed,
4. pointer-array,
5. tree,
6. chain,
7. circular-chain.

We did not include a hashing access structure in our design because it does not coexist well with other access

structures due to the special placement of the tuples in its range. Also, having it alone on a relation is not cost effective because the cost of processing most views except for selection is very high using a hashing access structure.

We need the following definitions in our discussion of the ranges of access structures.

> Definition 3.1 The range of a relation is the set of physical tuples that exists in the relation's extension.
>
> Definition 3.2 The range of an attribute is the same as the range of the relation on which the attribute is defined.

In our design, the ranges of all the simple access structures but those of the pointer array and circular chain types are equal to the ranges of the relations with which they are associated. This is due to the specific transformations we have in the metamorphosis. The range of a pointer array (or a circular chain) may be smaller than the range(s) of the attribute(s) with which it is associated.

In associating compound access structures with the logical database model, we use the term attribute access structure set. In an attribute access structure set, we

have more than one alternative pointer compound access structure. Only one of these compound access structures may be chosen in the optimization. The attribute access structure set as a whole represents all the simple access structures and hence the selection type of views that are associated with an attribute.

The compound access structures and the attribute access structure sets that we utilize are listed in the following:

1. cluster or tree aggregation of arrays,
2. cluster or tree aggregation of arrays and a tree generalization,
3. cluster or tree aggregation of arrays and a chain generalization,
4. pointer array generalization,
5. tree generalization,
6. chain generalization,
7. cluster generalization,
8. sorted generalization,
9. pointer array and tree generalizations,
10. pointer array and chain generalizations,
11. pointer array, tree and chain generalizations,

12. tree and chain generalizations,

13. well-placed generalization.

The properties of access structures include all the information needed of them and their constituent views by the optimizer. The main properties of an access structure are the list of the views that are represented by the access structure, total cost of processing these views and the size of the access structure's range.

In our design, we apply aggregation with pointer array and circular chain access structures whose ranges are smaller than those of the attributes with which they are associated. The ranges of other access structures are equal to that of the attribute. Also in choosing the range of generalizing access structures, we used the most common characteristics of the constituent access structures as far as data accessing is concerned.

First Set of Transformations

In these transformations, we transform each view into a set of simple access structures. Each access structure alternative is a good access structure to process the view under some global conditions (such as optimization constraints and the presence of other access structures). Since it is not possible to guess correctly the exact

condition that may occur in the optimal case, we include several alternative simple access structures to represent a view. In choosing the access structures commonly known algorithms to process the views are considered. The set of alternatives for each view is not exhaustive. In limiting the number and sort of alternative access structures corresponding to a view in the transformations, the optimization algorithm has been the major factor. A view is transformed into one or more simple access structures according to its type as shown in the following.

1. Intersection (R, S), union (R, S) and difference (R, S):

   i) A pointer array. The range of the pointer array contains tuples from R and S that have the same values for all the attributes. This access structure gives us the addresses of the physical tuples in the range of the view immediately.

2. Projection (R):

   i) Cluster on relation R. In obtaining the projection of R all the tuples in the relation R are accessed. Having all the tuples of R together by means of a cluster access structure minimizes the number of blocks that need to be accessed

3. Cartesian product (R, S):

   i) A cluster on R and another one on S. In

obtaining the cartesian product, all the tuples of both R and S are accessed. Having a cluster on R and another one on S minimizes the number of block accesses here also.

4. Quotient (R, S):

   i) A sorted access structure on R and another one on S on the first common attributes. Sorting R and S on their first common attribute expedites the comparison of the values of the tuples in both relations.

5. Selection (F, R):

   i) A pointer array with a range containing only the tuples in the range of the view. This access structure gives the addresses of the tuples in the range of the view immediately.

   ii) A pointer array with a range containing all the tuples in the relation. Each element of this pointer array contains a value of a tuple on the attribute related to the view and a pointer to the location of the tuple. This access structure eliminates the necessity of having to go through all the tuples in the relation.

   iii) A tree on R. A tree access structure gives relatively fast access to the tuples in the range of a selection view and is especially

usefull when the size of the view's range is small.

iv) A chain on R. This access structure is useful when the tuples of the relation are seperated in physical storage due to a well-placed access structure. This expedites the process of going through the whole relation associated with the selection view.

6. Join (H, R, S):

i) A circular chain containing only the tuples in the range of the view. By traversing the circular chain all the tuples in R that are joined to a tuple in S are obtained.

ii) A pointer array containing only the tuples in the range of the view. This access structure gives the addresses of the tuples that are joined.

iii) A well-placed on R and S (This is only for equi-join.). In this access structure each tuple in R is placed next to the tuples in S that it is joined. By going through the access structure once gives us the tuples in the range of an equi-join view.

Each simple access structure has the following properties associated with it:

i) the properties of the view it represents,

ii) the cost of processing the view it represents,

iii) the size of the access structure's range.

Costs

The total cost associated with an access structure in our model consists of the following costs

i) insertion cost, cinsert,

ii) deletion cost, cdelete,

iii) look-up cost, clook,

iv) consistency cost, cconsists,

v) storage cost, cstor,

vi) building cost, cbuild,

vii) constituent access structure search cost, cconstituent.

For specific access structure types some of the costs specified above may have zero values. When designing the detailed cost functions, adjustments should be made to each cost function so that all of the costs are in the same units.

Insertion, deletion and look-up costs are the total costs of processing the constituent views. Thus they are

determined by the views and the characteristics of the access structures. Consistency cost of an access structure is determined by the views that are not constituent views of the access structure and whose ranges coincide with that of the access structure. Consistency cost is needed because insertion and deletions of a view cause changes in the structure of the access structures that share the same range as the view. The building and storage costs are determined solely by the access structure. The constituent access structure cost is associated with using the aggregating access structure part of an aggregating compound access structure.

We can examine the total cost of an access structure x as a sum of the constituent view and access structure dependent costs, cview-dep, and the access structure dependent costs only, cstruct-dep, as shown in the following :

ctotal(x) = cview-dep(x) + cstruct-dep(x) .

Cost cview-dep contains the costs cinsert, cdelete and clook. Cost cstruc-dep contains cconsist, cstor, cbuild and cconstituent. However the individual costs contained in cview-dep and cstruct-dep may be zero depending on the access structure type.

The computation of cview-dep and cstruct-dep differ for simple access structures, generalization compound

access structures and aggregation compound access structures. For a simple access structure s that represents a view h the components of the total cost ctotal(s) are shown in the following.

$$\begin{aligned} \text{cview-dep}(s) = {} & \text{cinsert}(s, h) * \text{pi}(h) + \\ & \text{cdelete}(s, h) * \text{pd}(h) + \\ & \text{clook}(s, h) * \text{pl}(h) , \end{aligned}$$

where pi(h), pd(h) and pl(h) are the frequencies of insertions, deletions and look-ups associated with a view h, respectively. They are going to be used in computing the costs of other access structure types, too.

$$\text{cstruct-dep}(s) = \text{cbuild}(s) + \text{cstor}(s) + \text{cconsist}(s) ,$$

where

$$\begin{aligned} \text{cconsist}(s) = \sum_{\substack{k = 1, \\ k \neq s}}^{m} & (\text{cdelete}(s, k) * \text{pd}(k) + \\ & \text{cinsert}(s, k) * \text{pi}(k)) , \end{aligned}$$

m being the number of views that share the same range as access structure s.

For a generalization access structure g with n constituent access structures, the components of the total

cost ctotal(g) are computed as shown in the following.

$$\text{cview-dep}(g) = \sum_{i=1}^{n} (\text{cinsert}(g, v(i)) * pi(v(i)) + \text{cdelete}(g, v(i)) * pd(v(i)) + \text{clook}(g, v(i)) * pl(v(i))) ,$$

where v(i) gives us the view associated with simple access structure i.

$$\text{cstruct-dep}(g) = \text{cbuild}(g) + \text{cstor}(g) + \text{cconsist}(g) ,$$

where

$$\text{cconsist}(g) = \sum_{k=1}^{m} (\text{cdelete}(g, v(k)) * pd(v(k)) + \text{cinsert}(g, v(k)) * pi(v(k))) ,$$

where v(k), k (i, n), are the views that are not constituents of g, but whose ranges intersect the range of g.

For an aggregation access structure a with n constituent access structures, the components of the total cost ctotal(a) are computed as shown in the following.

$$\text{cview-dep } (a) = \sum_{i=1}^{n} (\text{cinsert}(i, v(i)) * pi(v(i)) + \text{cdelete}(i, v(i)) * pd(v(i)) + \text{clook}(i, v(i)) * pl(v(i))).$$

Let agg(a) give the aggregating access structure of an aggregation compound access structure a.

$$\text{cstruct-dep}(a) = \sum_{i=1}^{n} (\text{cbuild}(i) + \text{cstor}(i)) + \text{cbuild}(\text{agg}(a)) + \text{cstor}(\text{agg}(a)) + \text{cconstituent}(a) + \text{cconsist}(a) ,$$

where

$$\text{cconsist}(a) = \sum_{i=1}^{n} (\sum_{\substack{k=1, \\ k \neq i}}^{m(i)} (\text{cinsert}(i, k) * \text{pi}(k) + \text{cdelete}(i, k) * \text{pd}(k))) ,$$

m(i) giving the number of views that share the same range as access structure i,

and

$$\text{cconstituent}(a) = \sum_{i=1}^{n} (\text{clook}(\text{agg}(a), \text{vw}) * \text{pl}(v(i))).$$

Here vw indicates that look-up search is for one element, only.

Second Set of Transformations

The intermediate objects of the metamorphosis, i.e. the simple access structures, are transformed into compound access structures using the second set of transfor-

mations. According to view types, the simple access structures are transformed into compound access structures as follows.

A. Intersection(R, S), difference(R, S) and union (R, S): All the pointer arrays resulting from different views that are on the same R and S pair are homogeneously generalized into a pointer array that covers the same range as its constituent access structures but has a different cost property.

B. Projection(R) and cartesian product(R, S): All the cluster access structures on R and S resulting from projection(R) and cartesian product(R, S) are homogeneously generalized into a cluster access structure on R and S.

C. Quotient(R, S): All sorted access structures on the same relation are homogeneously generalized under a sorted access structure on the same relation.

D. Join(H, R, S): All the well-placed access structures on the same pair of relations resulting from equijoin are homogeneously generalized into a well-placed access structure on the same range as that of its constituents.

E. Selection(F, R): Selection being the most commonly used view, we have various choices for the compound access

structures that represent selections. Each compound access structure brings together simple access structures that represent views on the same attribute. This is because, in accessing, the attribute name is a frequently used common property of the selection views that brings about good grouping. Choosing another property of the views in bringing together access structures may bring about less efficient compound access structures and may be contrary to the goals of the compound access structure. For each attribute we have the following possible attribute access structure sets and compound access structures. Each of these compound access structures or attribute access structure sets represents the same set of views.

1. Generalization: In the following, heterogeneity and homogeneity make sense when we consider the fact that each view has a best representative simple access structure that takes part in the globally most optimal set of access structures (with respect to the alternatives provided). Although this simple access structure is one of the alternatives for a view, we do not know it exactly before the optimization.

a) Heterogeneous generalization:

i) Pointer array: All simple access structures that represent the views associated with an attribute are generalized into a pointer array whose range

is equal to that of the attribute.

ii) Tree: All simple access structures that represent the views associated with an attribute are generalized into a tree whose range is equal to that of the attribute.

iii) Chain: All simple access structures that represent the views associated with an attribute are generalized into a chain whose range is equal to that of the attribute.

b) Homogeneous generalization

i) Array and tree attribute access structure set: Array simple access structures are generalized into an array compound access structure and tree access structures are generalized into a tree access structure. Each compound access structure has a range equal to that of the the attribute with which it is associated. Depending on its insertion, deletion and look-up costs, each view is represented by either a tree or an array simple access structure whose range is equal to that of the view.

ii) Array and chain attribute access structure set: Array simple access structures are generalized into an array compound access structure and chain access structures are generalized into a

chain access structure. Each compound access structure has a range equal to that of the the attribute with which it is associated. Depending on its insertion, deletion and look-up costs, each view is represented by either a chain or an array simple access structure whose range is equal to that of the view.

2. Aggregation: In aggregation we generate the following attribute access structure sets:

i) cluster or tree aggregation of pointer arrays,

ii) cluster or tree aggregation of pointer arrays and a tree generalization of the rest of the access structures,

iii) cluster or tree aggregation of pointer arrays and a chain generalization of the rest of the access structures.

In generating these compound access structures, we first generate the cluster aggregation and if the number of the arrays aggregated is large, then we try tree aggregation to see if it reduces the total cost. In choosing the simple access structures to represent the views in ii) and iii), we consider the density defined in the following (Definition 3.3). But first we explain the representative and default access structures of a view.

The representative access structure is the access structure chosen during metamorphosis and the default access structure is the access structure used to process a view in case its representative is not chosen in the optimization. The representative and default access structures are shown according to the constituent view types in table 2.

<u>Definition 3.3</u> Profit, pf, of an access structure is the cost, c, of processing the set of views represented by the access structure subtracted from cost of processing the same set of views by the default access structure, d, i.e.

$$pf = d - c .$$

<u>Definition 3.4</u> Density of an access structure is defined as

$$density = profit / storage\ space\ requirement.$$

Since the default access structure is not known for the optimal case previously, it is an arbitrary value in the density function.

Before choosing the constituent array structures, we compute the following:

1. ca = cost of the access structure cluster aggregation of arrays,

2. sa = storage requirement of the access structure cluster aggregation of arrays,

3. ct = cost of processing all the views using a tree generalization,

4. st = storage requirement for all the views using a tree generalization.

Densities of cluster of array aggregation access structure, a, and the tree generalization access structure, t, are shown in the following. Let x stand for the default cost of processing the views. The density of access structure a is given by

$$da(x) = (x - ca) / sa .$$

The density of access structure t is given by

$$dt(x) = (x - ct) / st .$$

In the most likely case $ca < ct$, $sa > st$ and $-(ca/sa) > -(ct/st)$. The graphs of da(x) and dt(x) in such a case are shown in figure 6.

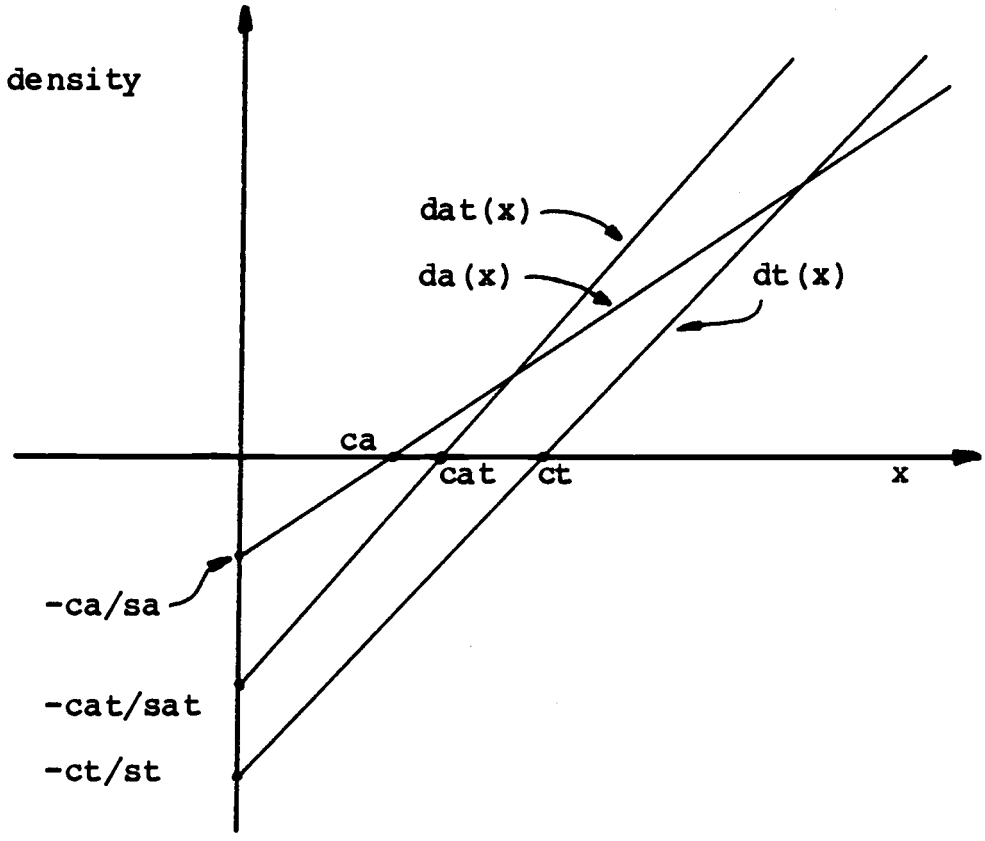


Fig. 6. Density functions with different cost and storage values.

In forming the aggregation access structures, we would like to have them with density functions with high values for most x values. This may be achieved if we keep cat close to ca and -cat/sat close to -ct/st. In general this means keeping cost and storage requirements as low as possible.

To achieve this goal at least partially, we transform the constituent pointer array access structures with (cat/sat) >= (ca+ct)/(st+sa) into a cluster or tree aggregation of pointer arrays. The rest of the simple access

structures are generalized into a tree or a chain access structure. The more of these constituent access structures we have with higher densities, the higher the slope of the resultant attribute access structure density will be. An attribute access structure set with a density function similar to dat(x) may be obtained only in cases where there are constituent simple access access structures with high densities.

IV. GLOBAL OPTIMIZATION OF THE ACCESS STRUCTURES

Metamorphosis gives us an efficient set of access structures that corresponds to a set of input views. However we can not and do not want to use all of these access structures together. We do not want to use them all because of redundancy in view representation, i.e. several access structures represent completely the same set of views. Furthermore, even if there were no redundancies, we could not use all of them because of space conflict of placement access structures and the storage space limitation imposed by the database system on the pointer type of access structures. Space conflict arises due to our assumption that a tuple of a relation may not be duplicated. Thus the ranges of two or more placement access structures may not coincide.

We specify the optimization problem that we tackle in the rest of this dissertation as follows.

<u>Optimization Problem</u>: Given a set of views and access structures that represent the views, choose a subset of the access structures that

i) minimizes the total cost of processing the views,

ii) does not cause any space conflicts and

iii) does not violate the space limitation.

In the solution set each view is represented by one access structure as i) above implies. However a view may be represented either by the representative access structure chosen by the metamorphosis or another access structure called the default access structure which is also generated by the metamorphosis but to represent another set of views. The possibility of default representation of a view makes the results more efficient at the expense of making the problem more difficult.

A subset of this optimization problem, namely considering only pointer type of access structures, is an NP-complete problem (see Ip[12]). This implies that our optimization problem is at least as hard as NP-complete problems. Since obtaining an exact solution to this problem is not practical, we give an approximate soluton to it.

An approach to solving this problem exhaustively is by considering each possible structure from the root to a leaf of the tree representation of the search space shown in figure 7.

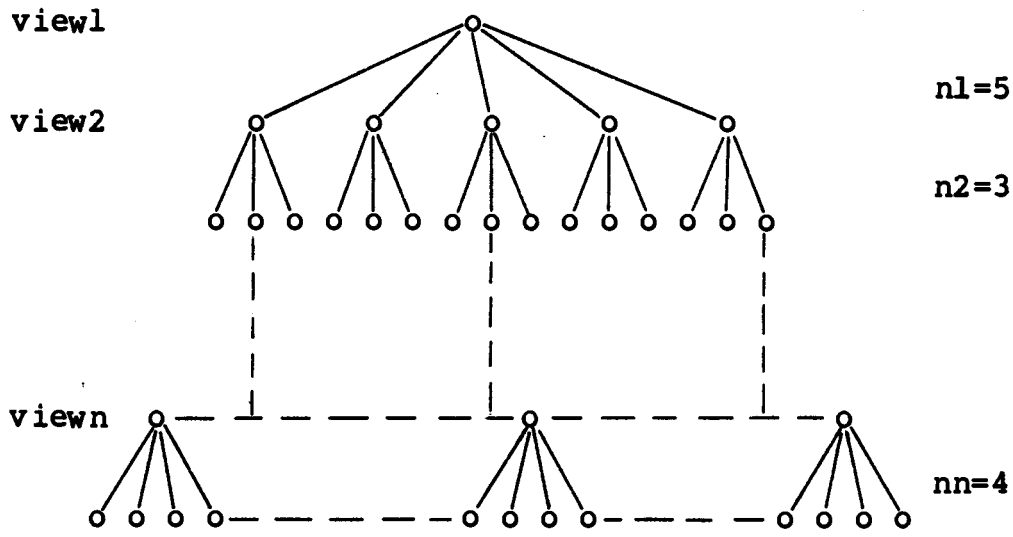


Fig. 7. The solution search space for n views.

In figure 7, we have n views and for each view we have nj alternative access structures to process the view. If the average number of alternatives for a view is k we have k**n choices to consider in the exhaustive search approach to obtain the exact solution.

In our approximate solution, we make use of separate placement and pointer access structure optimizations and combine them in a special way. Placement and pointer access structure optimizations are explained in the coming chapters. Also in our approach we have a different search space due to the fact that the input views are embedded in the access structures.

We divide the global optimization of access structures into placement and pointer access structure optimizations. When considered separately, each of these is used in obtaining optimal pointer or placement access structures. However in the global optimization of access structures the results obtained from pointer and placement access structure optimizations do not necessarily give the globally optimal set of access structures because of the interdependencies of the placement and pointer access structures as explained in chapter seven. Thus we give an algorithm to compensate for the dependencies.

We divide the problem into smaller parts because the presently available general approximate optimization techniques (i.e. polynomial time approximation techniques optimizing within a given bound) are not capable of effectively solving the overall optimization due to the complex nature of the problem. In our approach we categorize the access structures as placement and pointer access structures and then optimize them separately. The basis for this categorization is the minimality of the interdependence between the categories. That is the interdependence between placement and pointer access structures is relatively small. However the access structures in one category (placement or pointer) are tightly dependent on other structures of the same category, both

costwise and constraintwise. This division, in addition to helping us tackle the global optimization problem, provides alternative solutions to pointer and placement access structure optimization problems.

V. PLACEMENT ACCESS STRUCTURE OPTIMIZATION

Presentation of the Problem

In this chapter we examine the optimization of placement access structures separately. The optimization of placement access structures may be considered as a part of the total access structure optimization or on its own as the problem of structuring the tuples of a relational database with respect to linkage and selection operations. As a part of the global access structure optimization, it is used in achieving an approximate set of placement access structures as explained in chapter VII. On its own the problem is obtaining placement access structures of a relational database. In either case the difference between two different uses of the algorithm affects the costs and the view lists associated with the access structures.

The placement access structure optimization is defined as: given a set of placement access structures with weights representing the cost of processing the views the access structures represent, find a nonconflicting subset of these access structures that minimize the cost of processing the views. In a solution to this problem, there is only one access structure related to each rela-

tion. Thus when this problem is considered as a part of the general access structure optimization problem, the weight of each access structure represents the cost of processing all the views related to a relation.

The placement access structures of a relational database may be represented or modelled by a graph. The nodes of the graph represent the relations in the database; the self edges represent either cluster or sorted access structures associated with a relation; an edge between two nodes represents a well-placed access structure and a weight on an edge represents the cost of processing the views associated with an access structure. A graph representation of a set of placement access structures is given in the following example.

Example 5.1

A relational database with 5 relations and the following access structures may be represented pictorially by

a graph shown in figure 8, where the uncircled numbers represent the weights and the circled ones the id's.

sorted access structures:

relation name	weight
1	5
2	15
3	20
4	10
5	8

cluster access structures:

relation name	weight
1	10
2	10
3	30
4	15
5	5

well-placed access structures:

relation names	weight
1-2	15
1-3	13
1-4	8
1-5	10
2-4	20
2-5	18
3-4	25
4-5	10

In the optimization we consider only one self-edge with a node and one edge between any two nodes because we can locally eliminate the costlier of the two edges between the same nodes without effecting the global optimization's outcome. This elimination helps reduce the search space of the problem.

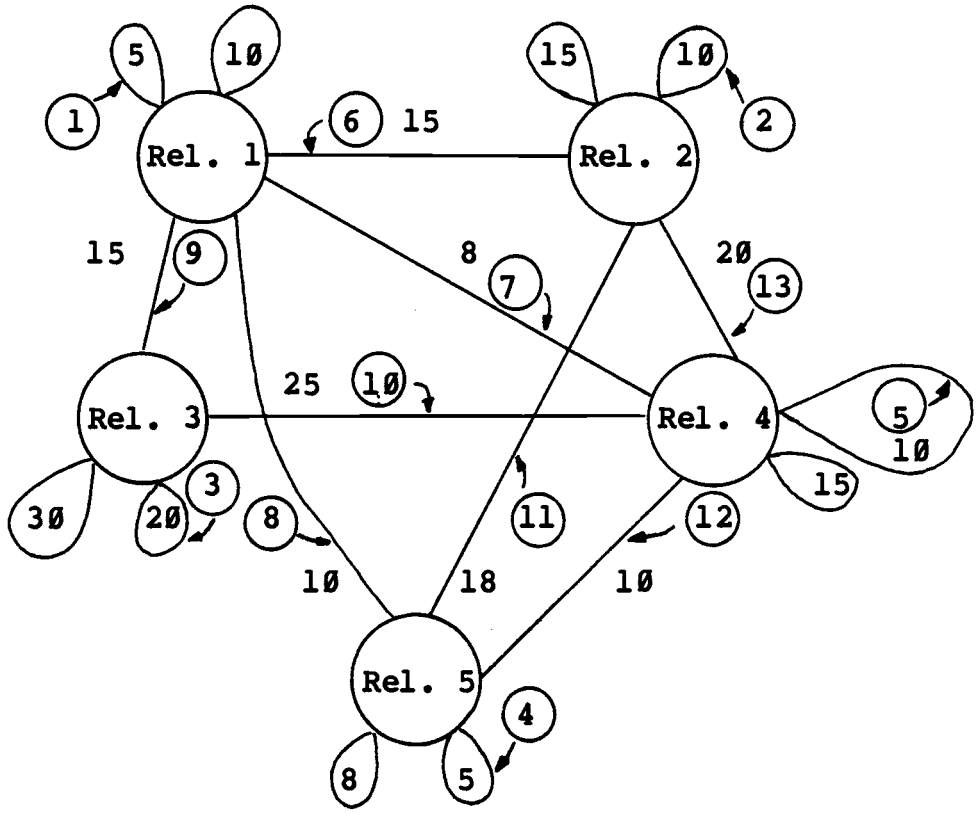


Fig. 8. Placement access structure modelling of a relational database.

We assume that the graph representing the placement access structures of a database is connected. If it is the case that the resulting graph is not connected, then each connected subgraph is independent of others as far as the optimization of placement access structures is concerned.

The statement of the placement access structure optimization problem in graph theoretical terms is as follows.

<u>Placement access structure optimization problem</u>:

Given a connected graph G=(V, E) with n nodes v_i, $i \in (1, n)$ and m edges e_j, $j \in (1,m)$ such that each edge, e_j, has a cost, $c(e_j)$ associated with it and that there is a self-edge associated with each node,

$$\text{minimize} \sum_{k=1}^{m} c(e_k) * a(k),$$

where $a(k)=\emptyset$ or 1, satisfying the condition that for each node v_i, $i \in (1,n)$, there is one and only one edge e_j with $a(e_j)=1$ such that $v_i \in FS(e_j)$, where $FS(e_j)=\{v_z, v_y\}$ for $j \in (1,m)$ and $z,y \in (1,n)$. FS gives the nodes associated with each edge.

This condition ensures that the tuples of each relation take part in the range of exactly one placement access structure and also that there is no space conflict among the edges. The space conflict for sorted, cluster and placement access structures associated with a database is defined in the following.

<u>Definition 5.1</u> Two edges e_p and e_r with $a(e_p)=a(e_r)=1$ are in conflict if and only if $FS(e_p) \cap FS(e_r)$ is non-empty.

A Solution to Placement Access Structure Optimization

We formulate the problem as a Ø-1 pure integer programming problem. The formulation of the problem for a graph with n nodes and m edges is given in the following.

$$\text{minimize} \sum_{i=1}^{m} c_i a_i ,$$

where $a_i = Ø$ or 1 and c_i is the cost of the access structure i and $c_i \geq Ø$, subject to the constraints: for each $k \in (1, n)$

$$\sum_{j \in m_k} a_j = 1 \quad ,$$

where m_k is the set containing the edges that are incident to node k.

The constraints ensure that there is one and only one edge associated with each relation.

Example 5.2

For the placement access structures marked with circled numbers in example 5.1 (which is redrawn in figure 9), we formulate the following Ø-1 integer problem.

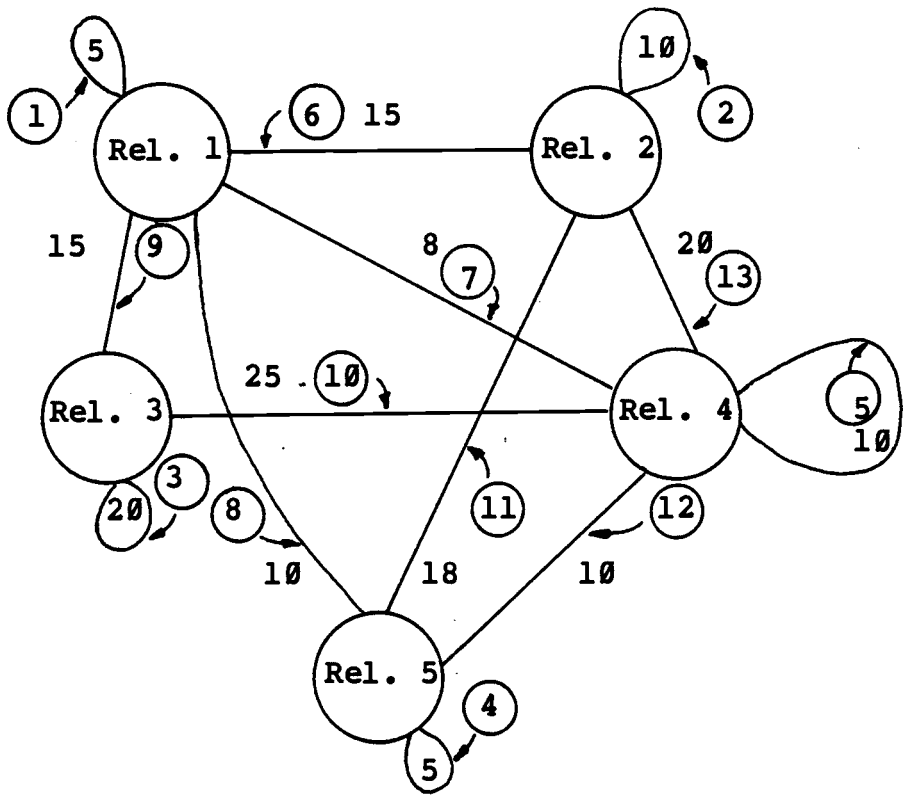


Fig. 9. Placement access structure model of the relational database in example 5.1.

minimize Z

$$Z = 5a_1 + 10a_2 + 20a_3 + 5a_4 + 10a_5 + 15a_6 + 8a_7 + 10a_8 + 15a_9 + 25a_{10} + 18a_{11} + 10a_{12} + 20a_{13}$$

subject to

$$a_1 + a_6 + a_7 + a_8 + a_9 = 1$$

$$a_2 + a_6 + a_{11} + a_{13} = 1$$

$$a_3 + a_9 + a_{10} = 1$$

$$a_5 + a_7 + a_{10} + a_{12} + a_{13} = 1$$

$$a_4 + a_8 + a_{11} + a_{12} = 1$$

The minimum cost placement access structure set consists of a_2, a_3 and a_{12}.

The 0-1 integer programming algorithm is a variety of branch and bound algorithm based on the branching and eliminating conditions of Balas[3].

Discussion

The placement access structure optimization is a difficult problem so as not to yield itself to polynomial solution methods that are in the literature such as rounding and interval partitioning and that give approximate solutions within a bound. This difficulty comes from the rigidity of the constraints of the placement access structure optimization problem. In other words it is difficult to get feasible solutions for the placement access structure optimization problem. In contrast in knapsack optimization problem (see Horowitz [9]) for example, any assignment of variables with total space requirement less then the total space capacity is feasible. In the placement access structure optimization, the assignments

obtained by incrementally considering the variables or the placement access structures do not yield very many feasible solutions. Thus it is not possible to apply a dominance relationship effectively and get rid of intermediate states if the problem is abstracted as a state space search.

Considering the relatively small search space of the problem, we think that zero-one integer programming formulation of the problem is appropriate. The Balas algorithm used in solving the integer programming problem has an exponential worst case time order but computational experience indicates that it has a fairly efficient average case behavior (see Hu [10]).

VI. POINTER ACCESS STRUCTURE OPTIMIZATION

Problem Statement

In this chapter, we consider the problem of selecting an optimal subset of the pointer access structures generated by the metamorphosis algorithm. According to our previously explained optimality criteria, we would like the chosen subset of pointer access structures to process all the views that are represented by the pointer access structures as optimally as possible. In this optimization, the secondary storage space is a constraint.

In metamorphosis each view is represented by one access structure, with the exception of selection type views. As for selection type views, metamorphosis generates several access structure sets for each set of selection views associated with an attribute. Obviously a necessary requirement for the optimization of pointer access structures is that each view is represented by at most one pointer access structure. To satisfy this requirement, we locally eliminate all but one of the attribute access structures associated with an attribute, using the dominance and density rules that are explained in the following.

Dominance Condition 1:

Let pointer access structures x and y represent the same set of views. Let p_x and p_y be the profits and s_x and s_y be the storage space requirements of x and y, respectively. If $p_x > p_y$ and $s_x \leq s_y$, then access structure x dominates y.

Dominance Condition 2:

Let c_x and c_y be the costs of processing the same set of views represented by access structures x and y, respectively. Let s_x, s_y, x and y be defined as in the dominance condition 1. If $c_x < c_y$ and $s_x \leq s_y$, then access structure x dominates y. This condition follows from the definition of profit and dominance conditon 1.

Dominance Rule:

If an attribute access structure set x dominates attribute access structure set y, we can eliminate y without effecting the outcome of the global access structure optimization.

We use the dominance rule in eliminating some of the attribute access structure sets associated with an attribute. However the dominance rule does not guarantee eliminating all but one attribute accesss structure set of an attribute. To eliminate all but one of the rest of the

attribute access structure sets, we make an approximation and use the density rule (explained in the following), after applying the dominance rule on the attribute access structure sets of an attribute.

<u>Density Rule</u>:

> If an attribute access structure set x has a higher density than the density of an attribute access structure set y, then eliminate y.

Pointer access structures with high densities are desirable because they occupy relatively less space for the amount of profit they provide.

In the resulting pointer access structure, S, of the metamorphosis after applying the dominance and density rules, each view is represented by only one access structure, assuming that ties among placement access structures with the same profit and space values are broken in favor of the access structure with the highest profit, if there are any ties. Thus choosing from S, the set of access structures that maximize the total profit also minimizes the total cost of processing the input set of views. We specify the optimization of pointer access structures as follows.

Pointer Access Structure Optimization Problem

Given n pointer access structures such that each view is represented by only one access structure,

$$\text{maximize} \sum_{i=1}^{n} p_i a_i$$

subject to the constraint

$$\sum_{i=1}^{n} s_i a_i \leq M ,$$

where p_i is the profit and s_i is the storage requirement of an access structure i ; M is the storage capacity of the system and a_i is equal to zero or one. It is assumed that

$$M < \sum_{i=1}^{n} s_i .$$

This problem is known as the Ø-1 knapsack problem.

The zero-one knapsack problem is an NP-complete problem (see Garey and Johnson [6]). Thus in the following section, we are going to use a polynomial time approximation algorithm to solve the pointer access structure optimization problem.

We would like to point out that secondary index selection in general may be stated as a Ø-1 knapsack problem. Hence the algorithm we use in the next section may be used for secondary index selection problems in general.

A Solution to The Pointer Access Structure Optimization Problem

We use the rounding technique in our approximation algorithm. The rounding method is used in obtaining polynomial time order approximate solutions to a certain class of optimization problems. It is based on transforming a problem instance into another which contains fewer objects. Thus the exhaustive search space is restricted and the resulting optimal solution is within a prespecified bound. The particular approximate algorithm that we use is by Ibarra [11]. The algorithm is also explained by Horowitz and Sahni [9].

The algorithm first finds a good upper bound for the optimal solution. Then using this upper bound the access structures are divided into two sets BIG and SMALL according to their densities. A dynamic programming exhaustive search algorithm is applied to BIG to find the set of feasible solutions. Then using the access structures in SMALL, an approximately optimal set of access structures is obtained. The algorithm is specified in detail in the following.

The pointer access structure optimization algorithm

1. Find an upper bound, UB:

a) Order n access structures such that

$$p_i/s_i \geq p_{i+1}/s_{i+1}, \quad 1 \leq i < n.$$

b) Find the largest j such that

$$\sum_{i=1}^{j} w_i \leq M.$$

c)
$$UB = \sum_{i=1}^{j+1} p_i.$$

2. Given a target tolerance ϵ, we divide n access structures into 2 classes BIG and SMALL such that we satisfy the inequality (optimal solution - approximate solution)/ optimal solution $< \epsilon$. BIG includes all objects with $p_i > \epsilon * UB/3$, SMALL includes the rest of the objects.

3. Round off the profit, p_i, of the r access structures that are in BIG. Replace each p_i by q_i such that $q_i = p_i/la$ where $la = UB * \epsilon^2/9$.

4. Apply the dynamic programming algorithm to the access structures in BIG using q_i's as the profit and leaving the constraints unchanged. In this algorithm the access structures are considered one by one starting from the first one. Basically this is an exhaustive enumeration algorithm. If we represent the

enumeration as a tree, the nodes are generated in a breadth first manner. At each stage (i.e. each time a new structure is considered) some of the structures are deleted using the dominance rule and the storage space constraint.

5. Let $S^{(r)}$ be the set of tuples obtained as a result of the dynamic programming algorithm. For each tuple (t_p, t_s), where t_p is the total profit and t_s is the total storage requirement of a feasible solution, fill the remaining space $M-t_s$ by considering the access structures in SMALL. The access structures in SMALL are considered in a nondecreasing order of density. After the filling, the tuple with the maximum profit is the answer.

6. The access structures whose profit and storage tuples constitute the chosen (t_p, t_s) in 5 are obtained by going back to states $S^{(r-1)}$, $S^{(r-2)}, \ldots S^{(0)}$.

The complexity of this algorithm is $O(n(\log n+1/\epsilon^2))$. Step one takes $O(n \log n + n)$ time. There are at most $UB/la = 9/\epsilon^2$ tuples in any $S^{(i)}$ (level i of the dynamic programming algorithm). There are r of these. Thus step 4 takes $O(r/\epsilon^2) < O(n/\epsilon^2)$. Step 5 takes $O(n/\epsilon^2)$ and step 6 takes $O(n)$ time.

The proof that this algorithm finds an approximation within a specified error bound makes use of the fact that $(1/2)*UB \leq P(I) < UB$, $P(I)$ being the optimal solution.

Example 6.1 illustrates the algorithm.

<u>Example 6.1</u>

Let us consider the following instance of the knapsack problem.

$(p_1, p_2, p_3, p_4, p_5) = (s_1, s_2, s_3, s_4, s_5) =$
(3, 5, 15, 100, 1000).

M = 1110.

$\epsilon = 1/10$.

step 1: Let the upper bound UB be equal to 1123.

step 2: $\epsilon * UB/3 = (1/10) * 1123 * (1/3) = 37$.

BIG = {(100, 100), (1000, 1000)}. These are the objects with p > 37.

SMALL = {(3, 3), (5, 5), (15, 15)}. These are the rest of the objects from the input.

step 3: $la = UB * \epsilon^2/9 = 1123 * 1/(100 * 9) = 1.2$.

BIG = {(83.3, 100), (833, 1000)}.

step 4: We apply the dynamic programming algorithm to the following instance of the knapsack problem: M=1110, n=2 and the profit-space pairs in set BIG from step 3.

$S^{(0)} = \{(0, 0)\}$, $S_1^{(0)} = \{(83.3, 100)\}$,

$S^{(1)} = \{(0, 0), (83.3, 100)\}$, $S_1^{(1)} = \{(833, 1000)\}$,

$S^{(2)} = \{(0, 0), (83.3, 100), (833, 1000),$
$(916.3, 1100)\}$.

step 5: Fill each tuple in S with elements from set SMALL observing the storage constraint M = 1110.

(0, 0) --> (23, 23)

(83.3, 100) --> (106.3, 123)

(833, 1000) --> (856, 1023)

(916.3, 1100) --> (924.3, 1108)

In the last one, we can not add all the elements in SMALL without violating the storage constraint.

The resultant access structure set has a profit of 924.3 and requires a storage space of 1108.

step 6: (916.3, 1100) is obtained in $S^{(2)}$ so (833, 1000) is in it. 916.3 - 833 = 83.3 and 1100 - 1000 = 100. This shows that (83.3, 100) is in the optimal set. The resultant optimal set contains 1, 2, 4 and 5.

VII. OVERALL ACCESS STRUCTURE OPTIMIZATION ALGORITHM

Introduction

In this chapter we present the overall access structure optimization algorithm. This algorithm considers the interdependencies between pointer and placement access structure optimization algorithms. Pointer and placement access structures are dependent on each other due to the costs and profits of access structures. Before we explain the interdependencies, we give the representative and default access structures according to the constituent view types in table 2.

Table 2. The default and representative access structures according to view types.

view	representative	default
quotient	placement	another placement
projection	placement	another placement
cartesian product	placement	another placement
selection	pointer	a placement
join	pointer	a placement
equi-join	a placement	another placement
intersection, difference and union	pointer	a placement

The pointer access structure optimization depends on the outcome of the placement access structure optimization because of the following dependency.

The profit of a pointer access structure depends on the placement access structure that covers the tuples that are in the range of the pointer access structure. This dependency is due to the following reasons.

i) The default cost (of processing the views) that is used in the profit computation depends

on the placement access structure that acts as the default.

ii) The cost of processing the views that constitute a pointer access structure depends on the placement of the tuples that are in the range of the pointer access structure, i.e. the placement access structures that cover the range of the pointer access structure.

The placement access structure optimization is dependent on the outcome of the pointer access structure optimization because of the default processing of the views whose pointer access structures are not chosen in the optimization.

If we knew the optimal placement (pointer) access structure set for a database, then we could compute the optimal set of pointer (placement) access structure set using the pointer (placement) access structure optimization algorithm explained previously. Unfortunately this is not possible because the pointer and placement access structure optimizations are circularly dependent on each other as shown in figure 10.

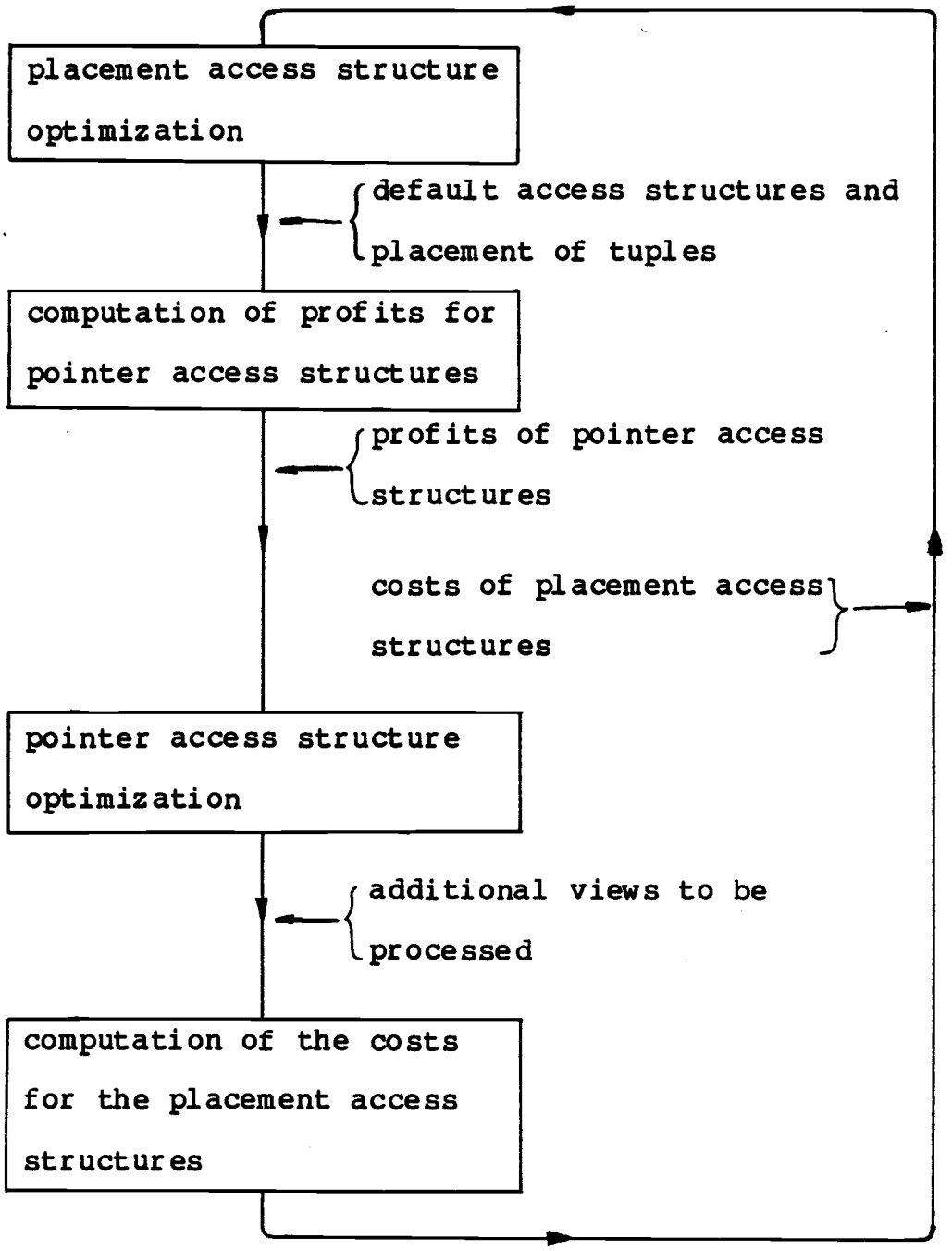


Fig. 10. Dependencies of placement and pointer access structure optimization algorithms.

To find the optimal access structure set of a database, we need to consider exhaustively each possible placement configuration (a placement configuration being a nonconflicting set of placement access structures that cover the whole database) and for each placement configuration compute the optimal pointer access structure set using the pointer access structure optimization algorithm. The number of possible alternatives makes such an exhaustive algorithm impractical to compute. Instead we make an approximation and try to get close to the optimal solution. In our approximation algorithm, explained in the following, we make use of the previously explained placement and pointer access structure optimization algorithms.

The Overall Access Structure Optimization Algorithm

1. Assume that each relation in the database is assigned either a sorted or a cluster placement access structure.

   a) Compute the profits of the pointer access structures that are in the range of the metamorphosis.

2. Assume that no pointer access structures are eliminated.

a) Compute the optimal set of placement access structures using the placement access structure optimization algorithm with the costs computed according to the assumption.

b) Compute the profits of pointer access structures based on the placement access structures chosen at step 2.a.

3. Let p_{j1} and p_{j2} be the profits of pointer access structure j computed at step 1.a and 2.b, respectively.

a) Compute the profits p for all access structures using the formula

$$p_j = (p_{j1} + p_{j2}) / 2 .$$

b) Compute the optimal set of pointer access structures using the pointer access structure optimization algorithm and the profits from 3.a.

REPEAT

4. a) Compute the costs of placement access structures considering the views whose representative pointer access structures are not chosen at step 3.

b) Compute the optimal set of placement access structures using the costs from 4.a and the

placement access structure optimization algorithm.

5. a) Compute the profits of the pointer access structures using the placement access structures obtained at 4.b.

   b) Compute the optimal set of pointer access structures using the pointer access structure optimization and the costs from 5.a.

UNTIL either

i) one of the placement or pointer access structure configurations is the same as the previous one or

ii) the difference in net costs between the last or the previous iterations (i.e. repetition of steps 4 and 5) is less then a certain percentage.

The access structures computed at step 4.b and 5.b constitute the approximately optimal access structures of the database.

Rationale for the Overall Access Structure Optimization Algorithm

In our algorithm we first try to find the approximately optimal pointer access structure configuration of the database. It is important to have an idea about the views whose representative pointer access structures will not be chosen before computing the approximate costs of the placement access structures. The effect of these views on the costs of the placement access structures may be very significant. On the other hand the effect of the default costs on the profits are relatively small, i. e., only a part of each profit computation is dependent on the default access structure. After approximating the optimal pointer configuration, we find the optimal placement access structure configuration that is based on it.

In steps 1 to 3, we approximately find the views that will not be processed by their representative pointer access structures in the optimal pointer access structure set. In step 4, we compute the approximately optimal placement access structure set. In step 5, we compute the optimal pointer access structure set corresponding to the placement access structures computed at step 4. In the following paragraphs, we are going to compare some of the costs, profits and the access structures computed with those of the optimal configuration. Before we do that we

state the observations that are used in the rest of this section.

<u>Observation 7.1</u> :

It costs less to process views whose representative access structures are of pointer type using sorted and/or cluster access structures as defaults rather than well-placed access structures.

<u>Observation 7.2</u> :

The cost of default processing of a view is the determining factor in profit comparisons. This is due to the cost of default processing of a view being more than the cost of accessing the tuples that are in the range of the view when the same default placement access structure is used.

In step one we assume that there are no well-placed access structures in the placement configuration. However it is possible and likely that there are some well-placed access structures in the optimal placement access structure configuration. Considering this possibility and using observations 7.1 and 7.2, we deduce that the profits of pointer access structures computed at step one are lower or equal to those computed using the optimal placement access structure configuration. The profits computed at step one are lower bounds for those based on the

optimal placement access structure configuration.

In step 2.a, we assume that no views with pointer type of representative access structures are going to be processed by the placement access structures. The costs computed using this assumption cause more well-placed in general and more specifically well-placed with larger ranges to be chosen by the optimizer compared to the optimal placement configuration. To see this let us consider a node x and the placement access structures a, b and c shown in figure 11.

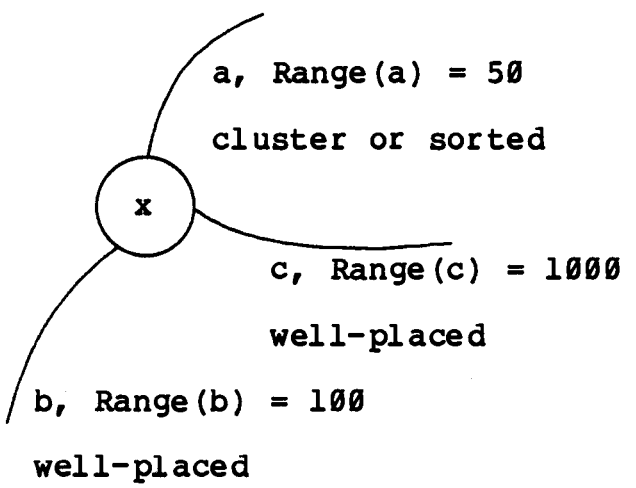


Fig. 11. Placement access structures with different ranges incident at a node.

The cost of processing any additional views that might have to be considered in the optimal configuration is the highest for c, less so for b and the least for a, as indicated by observation 7.1. The cost of processing additional views may cause c not to be chosen for the

optimal placement configuration. At step 2, however, due to the assumption, c has a better chance of being chosen because the differences in the costs of a, b and c are smaller.

Because of the likelihood of more well-placed access structures and well-placed access structures with larger ranges being chosen at step 2.a, the profits computed at step 2.b might be higher compared to those based on the optimal placement configuration. This follows from observations 7.1 and 7.2. The profits computed here are upper bounds. The approximate effect of the well-placed access structures, in the optimal placement configuration, on the profits is introduced here.

At step 3, we compute the profits that are between the upper and the lower bounds. Then based on these profits, we get the views whose representative pointer access structures are not chosen.

At step 4, we use these additional views computed at step 3 to get a better estimation of the costs of the placement access structures. As a result the placement access structures chosen are closer to the optimal set of placement access structures then those of step 2.

At step 5, we use the placement access structures chosen at step 4 to compute the pointer access structures.

In the algorithm the main approximation comes from steps 1 and 2. Step 1 gives the relative profits of the pointer access structures without regarding the effect of the underlying optimal placement configuration. Step 2 approximates the effect of the underlying placement configuration. This changes the relative profits of the pointer access structures. Then in step 4 the placement access structures are approximated, after getting a good pointer access structure configuration approximation in the previous steps. In step 5 the pointer access structures are computed again to fine tune them. Iteration of steps 4 and 5 helps if the fine adjustment at step 5 causes any changes in the pointer access structure configuration.

VIII. DISCUSSION

In this dissertation, we presented algorithms and concepts to automate the design of data structures in general and access structures for relational databases in particular. The results of this dissertation are not only helpful in designing more efficient and precise access structures to meet the demands and requirements of the users but, at the same time, the concepts introduced make the comprehension and analysis of complex data structures easier thus hopefully making the future design and automatic generation of complex data structures easier, too.

We presented the concepts of aggregation and generalization in access structure design and analysis and metamorphosis for generating access structures from views. In metamorphosis we modelled the user interactions and their usage patterns of the database in a rigorous way. We examined the global optimization of access structures as a selection of access structures to optimize processing the user views and showed the intrinsic problems and dependencies that make the problem a global and complex one. We gave an approximate algorithm to solve the global optimization problem. We gave one approximate algorithm to solve the pointer access structure optimization problem

and formulated the placement access structure optimization as an integer programming problem. We coded and tested the algorithms.

The whole design and the algorithms model the access structures, user interactions with the access structures and the underlying system associated with the access structures. A model is an abstraction and in an abstraction some of the details are omitted. However we feel that in our model we included enough details to make it a realistic one.

In our model we classified the data structures and showed precisely how any complex data structure may be obtained from the simplest data items. In our classification we considered most if not all of the presently known access structures. We did not limit the design to one or two access structure types as is the trend in the literature but considered various access structures with varying ranges. For we think at least in some implementations various access structures are used, system R [2] being one example.

Our conjecture on complex access structures is a unifying conceptual framework. It unifies the analysis and/or design of all complex access structures under one framework. It makes the automatic design of complex access structures easier and thus the generation of

special purpose and more efficient access structures possible.

We considered various costs to realistically model the usage of the access structures and the importance of each view. We refrained from utilizing grossly approximate and general cost formulas for the costs are highly system dependent and their accuracy has a crucial effect on the outcome of the access structure design.

In transforming users' usage patterns to access structures we made sure that the usage patterns are well reflected. In formulating the optimization problem, we considered the most important aspects of the system's efficiency criterion. In solving the optimization problem we tried as much as possible to stick to using exact or boundedly approximate solution methods.

In a way we showed that within the boundaries of the presently available general optimization methods, it is still possible to automate complex access structure designs. However as the general optimization methods improve the concepts developed in this dissertation will have a more significant impact and it will be possible to automatically design more precise access structures to suit the needs of the users. More specifically in our algorithm we would be able to represent a view with more access structures and defaults and have generally more

accurate global optimizations.

In the design methodology given here some of the general decisions -- such as the choice of access structure types to be used and the specifications of transformation rules -- may be made by the human designer using his/her expertise under the guidance of the concepts explained. However tedious decisions and procedures, where human beings are likely to err due to the large number of decisions and dependencies, are automated. In this way the results obtained from actual design reflect the skills of the designer. Thus we would expect the access structures built by our methodology to be at least as good as those of a human being. Moreover the automation of access structure design may be more cost effective or a necessity in some applications such as adaptive access structure design.

As for related future research directions, including multiattribute access structures in the metamorphosis is an open problem. Having multiattribute access structures increases the amount of dependencies and makes the optimization more complicated. Also it would be good to see the metamorphosis and the related concepts applied to other high level data models such as semantic data models which are becoming more popular.

BIBLIOGRAPHY

(1) Anderson, H. D., and Berra, P. B., Minimum Cost Selection of Secondary Indexes, ACM TODS, vol. 2, no. 1, March 1977, pp. 68-90.

(2) Astrahan, M., M., Blasgen, M., W., Chamberlin, D., D., Eswaran, K., P., Gray, J., N., Griffiths, P., P., King, W., F., Lorie, R., A., McJones, P., R., Mehl, J. W., Putzolu, J., R., Traiger, I., L., Wade, B., W., and Watson, V., System R: Relational Approach to Database Management, ACM TODS, vol. 1, no. 2, June 1976, pp. 97-137.

(3) Balas, E., An Additive Algorithm for Solving Linear Programs with Zero-One Variables, Operations Research, vol. 13, 1965, pp 517-546.

(4) Chang, J. M. and Fu, K. S., Extended K-d Tree Database Organization: A Dynamic Multiattribute Clustering Method, IEEE Trans. on Software Engineering, vol. SE-7, no. 3, May 1981, pp 284-290.

(5) Furukawa, K., Research Direction In the Very Large Database Area, Proc. Very Large Data Bases 3rd Int. Conf., Tokyo, Japan, October 6-8, 1977, p 169.

(6) Garey, R.M. and Johnson, D.S., Computers and Intractability ,W. H. Freeman and Company, 1979.

(7) Haerder, T., Implementing a Generalized Access Path Structure for a Relational Data Base System, ACM TODS, vol. 3, no. 3, September 1978, pp 285-298.

(8) Hatzopoulos, M. and Kollias, J. C., On the Optimal Selection of Multilist Database Structures, IEEE Trans. on Soft. Eng'g., vol. SE-10, no. 6, November 1984, pp. 681-687.

(9) Horowitz, E. and Sahni, S., Fundamentals of Computer Algorithms ,Computer Science Press, Inc., Computer Software Engineering Series, 1978

(10) Hu, T. C. and Robinson, S. M., Mathematical Programming ,Academic Press, 1973.

(11) Ibara, O. and Kim, C., Fast Approximation Algorithms for the Knapsack and Sum of Subsets Problems, JACM, vol. 22, no.4, October 1975, pp. 463-468.

(12) Ip, Y. L. M., Saxton, L. V., and Raghavan, V. V., On the Selection of an Optimal Set of Indexes, IEEE Trans. on Soft. Eng'g., vol. SE-9, no. 2, March 1983, pp. 135-143.

(13) Katz, R. H., and Wong, E., Resolving Conflicts in Global Storage Design through Replication, ACM TODS, vol. 8, no. 1, March 1983, pp. 110-135.

(14) Lipton, R. J., Rosenberg, A. L. and Yao, A. C., External Hashing Schemes for Collections of Data

Structures, Journal of ACM, vol. 27, no. 1, January 1980, pp 81-95.

(15) March, S. T., Techniques for Structuring Database Records, Computing Surveys, vol. 15, no. 1, March 1983, pp. 45-79.

(16) Roussopoulos, N., The Logical Access Path Schema of a Database, IEEE Trans. on Soft. Eng'g, vol. SE-8, no. 6, November 1982, pp 563-573.

(17) Roussopoulos, N., View Indexing in Relational Databases, ACM TODS, vol. 7, no. 2, June 1982, pp. 258-290.

(18) Rowe, L. A. and Tonge, F. M., Automating the Selection of Implementation Structures, IEEE Trans. on Soft. Eng'g., vol. SE-4, no. 6, November 1978, pp. 494-506.

(19) Scheuermann, P., On the Design and Evaluation of Data Bases, vol. 11, no. 2, February 1978, pp. 46-55.

(20) Schkolnick, M., Secondary Index Optimization, Proc. Int. Conf. Management of Data , San Jose, Calif., May 14-16, ACM, New York, 1975, pp. 186-192.

(21) Smith, J. M. and Smith, D. C. P., Database Abstractions: Aggregation and Generalization, ACM TODS, vol. 2, no. 2, June 1977, pp 105-133.

(22) Smith, J. M. and Smith, D. C. P., Database Abstractions: Aggregation, Communications of the ACM, vol. 20, no. 6, June 1977, pp 405-415.

(23) Tsichritzis, D. C. and Lochovsky, F. H., Data Models , Prentice Hall Inc., 1982.

(24) Ullman, J. D., Principles of Database Systems , Chapter 2,(Query Optimization), Computer Science Press, 1980, pp. 211-240.

(25) Ullman, J. D., Principles of Database Systems ,Computer Science Press, 1980.

(26) Yao, S. B., Waddle, V. E., and Housel, B. C., View Modelling and Integration Using The Functional Data Model, IEEE Trans. on Soft. Eng'g, vol SE-8, no. 6, November 1982, pp. 544-553.

APPENDIX

TWO SAMPLES OF ACCESS STRUCTURE GENERATION AND OPTIMIZATION

In the following, we give two examples to access structure generation and optimization, the first one being very simple and the second one is less so. In each sample, we first give a set of views of a relational database and then the access structures generated by the metamorphosis program. We then show the profits and/or costs of the access structures at each step of the overall optimization algorithm. We finally give the optimal access structures chosen for the input set of views. We would like to point out that the cost functions used in the optimization and metamorphosis programs are purely arbitrarily chosen. Also the abbreviations used are common to both samples.

abbreviations used :

v : view type, (permitted tyes are : p = projection, c = cartesian product, q = quotient, j = join, s = selection, i = intersection, d = deletion, u = union)

rl1 : relation one

rl2 : relation two

atl : attribute one

at2 : attribute two

size : size of the view's range

pl : frequency of look-ups

pi : frequency of insertions

pd : frequency of deletions

op : comparison operator

I. A VERY SIMPLE SAMPLE

The following example is given for illustrative purposes only and does not reflect all the features of the metamorphosis and the optimizations due to its undue simplicity. Let us assume that we have the following relational database schema and its instance.

Relation 1. Customers(name,city)
Relation 2. Parts(part#,price)
Relation 3. Orders(name,part#,quantity)

Physical Relations:

Customer:

Name	city
Earl Ecklund	Beaverton
John Byrnne	Corvallis
Bud Clark	Portland
Sam Earle	Beaverton
Ronald Reagan	Washington

Parts:	Part#	Price
	2c	$1000
	318i	$20000
	68000	$130
	309J	$50
	512k	$495

Orders:	Name	Part#	Quantity
	Earl Ecklund	309J	2
	John Byrnne	318i	3
	Earl Ecklund	68000	20
	Bud Clark	309J	50
	Bud Clark	2c	1
	Bud Clark	68000	1
	Sam Earle	2c	5
	Bud Clark	318i	1
	Sam Earle	512k	15
	Earl Ecklund	512k	18

Also let us assume that we have the following frequently used views that we want to use to build the access structures for the database.

View 1. Select (city='Corvallis', Customer)
View 2. Select (part#='309J', Parts)
View 3. Select (part#='318i', Parts)
View 4. Select (part#='512k', Parts)
View 5. Select (quantity>15, Orders)
View 6. Join (Customers, Orders, C.name=O.name)
View 7. Join (Parts, Orders, P.part#=O.part#)
View 8. Projection (name, Customer)
View 9. Projection (part#, Parts)
View 10. Projection (name, Orders)
View 11. Quotient (Orders, Customer)

1. INPUT VIEWS

views :

v	rl1	rl2	at1	at2	size	p1	pi	pd	op
p	1				5	1.0	1.0	1.0	
j	1	3	1	1	8	10.0	10.0	10.0	EQ
q	1	3			5	1.0	1.0	1.0	
p	2				5	1.0	1.0	1.0	
j	2	3	1	2	10	10.0	10.0	10.0	EQ
p	3				10	1.0	1.0	1.0	
s	1		2		1	40.0	4.0	8.0	EQ
s	2		1		1	60.0	80.0	8.0	EQ
s	2		1		1	100.0	150.0	100.0	EQ
s	2		1		1	40.0	30.0	60.0	EQ
s	3		3		3	120.0	10.0	30.0	GT

number of relations : 3

size of relations

rel : 1 size : 5
rel : 2 size : 5
rel : 3 size : 10

arity of relations

rel : 1 number of attributes : 2
rel : 2 number of attributes : 2
rel : 3 number of attributes : 3

2. CANDIDATE ACCESS STRUCTURES GENERATED BY THE METAMORPHOSIS

The value of "number of views" is the number of constituent views of the access structure.

A well-placed access structure is shown at both of its associated relations.

"poarray" stands for pointer array access structure.

<u>placement access structures</u>

```
relation :          1

  cluster
    total cost :  1.90260070411180e+04
    number of views :          1

  sorted
    total cost :  8.02296953778455e+03
    number of views :          1

relation :          2

  cluster
    total cost :  3.66250000000000e+02
    number of views :          1

relation :          3

  cluster
    total cost :  6.84263279555968e+04
    number of views :          1

  sorted
    total cost :  1.93472879252389e+04
    number of views :          1
    start add. :          7

relation :   1

  wellplaced  no. :          1
    total cost :  3.05978047044482e+03
    number of views :          1
    relations :          1,          3
```

```
relation :   2

  wellplaced  no. :          1
    total cost :  2.0493999788Ø398e+Ø3
    number of views :          1
    relations :          2,         3

relation :   3

  wellplaced  no. :          1
    total cost :  3.Ø5978Ø47Ø44482e+Ø3
    number of views :          1
    relations :          1,         3

  wellplaced  no. :          2
    total cost :  2.0493999788Ø398e+Ø3
    number of views :          1
    relations :          2,         3
```

pointer access structures :

access structure sets generated for attributes by selection views :

For each attribute there are eight separate access structure sets as explained on page 58f.

In the following the information given under "associated views" is the information about the constituent views. More specifically, "view address" gives the location of the view in the input view set and "strc. no." shows in which compound access structure in the attribute access structure set the view takes part. For example, "strc. no. 2" in an array tree access structure set means that the view is a constituent of

the second access structure in the attribute access structure set (i.e. tree).

```
rel :          1    attribute :          2

  tree
    cost :  3.99334826351515e+01
    storage :          2
    relation 1 :          1
    number of views :          1
    attribute 1 :          2

  poarray
    cost :  6.87500000000000e+01
    storage :          2
    relation 1 :          1
    number of views :          1
    attribute 1 :          2

  chain
    cost :  1.54100000000000e+01
    storage :          2
    relation 1 :          1
    number of views :          1
    attribute 1 :          2

  aggre. of poarray
    cost :  1.40200000000000e+01
    storage :          1
    relation 1 :          1
    number of views :          1
    attribute 1 :          2
```

```
array and tree
  cost :  4.36834826351515e+01
  storage :          4
  relation 1 :          1
  number of views          1
  attribute 1 :          2
  associated views :

    view address    8,    strc. no   2

array and chain
  cost :  1.91600000000000e+01
  storage :          4
  relation 1 :          1
  number of views          1
  attribute 1 :          2
  associated views :

    view address    8,    strc. no   2

aggr. of arrays and tree
  cost :  3.99334826351515e+01
  storage :          2
  relation 1 :          1
  number of views          1
  attribute 1 :          2
  associated views :

    view address    8,    strc. no   2

aggr. of arrays and chain
  cost :  1.74300000000000e+01
  storage :          3
  relation 1 :          1
  number of views          1
  attribute 1 :          2
  associated views :

    view address    8,    strc. no   1

rel :          2    attribute :          1

tree
  cost :  1.02083242297676e+03
  storage :          2
  relation 1 :          2
  number of views :          3
  attribute 1 :          1
```

```
poarray
  cost :  8.14150000000000e+02
  storage :          2
  relation 1 :          2
  number of views :          3
  attribute 1 :          1

chain
  cost :  4.51730000000000e+02
  storage :          2
  relation 1 :          2
  number of views :          3
  attribute 1 :          1

aggre. of poarray
  cost :  1.60060000000000e+02
  storage :          3
  relation 1 :          2
  number of views :          3
  attribute 1 :          1

array and tree
  cost :  8.63181279471776e+02
  storage :          4
  relation 1 :          2
  number of views          3
  attribute 1 :          1
  associated views :

    view address    9,    strc. no   1
    view address   10,    strc. no   1
    view address   11,    strc. no   1

array and chain
  cost :  4.80880000000000e+02
  storage :          4
  relation 1 :          2
  number of views          3
  attribute 1 :          1
  associated views :

    view address    9,    strc. no   2
    view address   10,    strc. no   2
    view address   11,    strc. no   2
```

```
aggr. of arrays and tree
  cost :  1.02083242297676e+03
  storage :          2
  relation 1 :          2
  number of views          3
  attribute 1 :          1
  associated views :

    view address    9,    strc. no   2
    view address   10,    strc. no   2
    view address   11,    strc. no   2

aggr. of arrays and chain
  cost :  4.51730000000000e+02
  storage :          2
  relation 1 :          2
  number of views          3
  attribute 1 :          1
  associated views :

    view address    9,    strc. no   2
    view address   10,    strc. no   2
    view address   11,    strc. no   2

rel :          3    attribute :          3

tree
  cost :  2.88413513220577e+02
  storage :          3
  relation 1 :          3
  number of views :          1
  attribute 1 :          3

poarray
  cost :  4.09700000000000e+02
  storage :          3
  relation 1 :          3
  number of views :          1
  attribute 1 :          3

chain
  cost :  4.57400000000000e+01
  storage :          3
  relation 1 :          3
  number of views :          1
  attribute 1 :          3
```

```
aggre. of poarray
  cost :  1.21020000000000e+02
  storage :          1
  relation 1 :          3
  number of views :          1
  attribute 1 :          3

array and tree
  cost :  2.98113513220577e+02
  storage :          6
  relation 1 :          3
  number of views          1
  attribute 1 :          3
  associated views :

    view address   12,    strc. no   2

array and chain
  cost :  5.54400000000000e+01
  storage :          6
  relation 1 :          3
  number of views          1
  attribute 1 :          3
  associated views :

    view address   12,    strc. no   2

aggr. of arrays and tree
  cost :  1.35101108898315e+02
  storage :          4
  relation 1 :          3
  number of views          1
  attribute 1 :          3
  associated views :

    view address   12,    strc. no   1

aggr. of arrays and chain
  cost :  1.26760000000000e+02
  storage :          4
  relation 1 :          3
  number of views          1
  attribute 1 :          3
  associated views :

    view address   12,    strc. no   1
```

3. MINIMUM COST SORTED OR PLACEMENT TYPE ACCESS STRUCTURE CONFIGURATION

rel : 1, type : sorted

rel : 2, type : cluster

rel : 3, type : sorted

4. PROFITS OF THE POINTER ACCESS STRUCTURES FOR THE PLACEMENT CONFIGURATION OF STEP 3

abbreviations :

intpo : data structure storing pointer access structures generated by inter, diff and union views.

joinpo : data structure storing pointer access structures generated by join views.

selecpo : data structure storing pointer access structures generated by selection views.

The last two entries for each access structure give us the location of the access structure in intpo, joinpo or selecpo. For example,

index to intpo, joinpo or selecpo : 3
type of structure : int

implies that the access structure is the third element of intpo.

pointer access structures in sorted order :

structure : 1
density : 1.25313731025000e+07
space : 2
profit : 2.50627462050000e+07
index to intpo, joinpo or selecpo : 2
type of structure : selecs

```
structure :          2
density :  6.49891774241899e+06
space :          1
profit :  6.49891774241899e+06
index to intpo, joinpo or selecpo :          3
type of structure : selecs

structure :          3
density :  8.59725792433009e+05
space :          1
profit :  8.59725792433009e+05
index to intpo, joinpo or selecpo :          1
type of structure : selecs
```

5. MINIMUM COST PLACEMENT ACCESS STRUCTURE CONFIGURATION WITHOUT CONSIDERING EXTRA VIEWS

rel : 1, type : well-placed with rel. 3

rel : 2, type : cluster

rel : 3, type : well-placed with rel. 1

6. PROFITS OF POINTER ACCESS STRUCTURES USING THE PLACEMENT CONFIGURATION OF STEP 5.

pointer access structures in sorted order :

```
structure :          1
density :  1.25313731025000e+07
space :          2
profit :  2.50627462050000e+07
index to intpo, joinpo or selecpo :          2
type of structure : selecs
```

```
structure :          2
density :  3.14747792050000e+07
space :          1
profit :  3.14747792050000e+07
index to intpo, joinpo or selecpo :          3
type of structure : selecs

structure :          3
density :  7.67441413000000e+06
space :          1
profit :  7.67441413000000e+06
index to intpo, joinpo or selecpo :          1
type of structure : selecs
```

7. AVERAGE OF THE PROFITS FROM STEPS 6 AND 4

Pointer access structures are listed in the order they appear in steps 4 and 6.

```
pointer  struct          1
profit from step 6 :  2.50627462050000e+07
profit from step 4 :  2.50627462050000e+07
average profit  2.50627462050000e+07

pointer  struct          2
profit from step 6 :  3.14747792050000e+07
profit from step 4 :  6.49891774241899e+06
average profit  1.89868484737095e+07

pointer  struct          3
profit from step 6 :  7.67441413000000e+06
profit from step 4 :  8.59725792433009e+05
average profit  4.26706996121650e+06
```

8. THE RESULT OF THE POINTER ACCESS STRUCTURE OPTIMIZATION

space limit : 3

approximation tolerance : 1.00000000000000e-02

number of pointer access structures : 3

abbreviations :

id : 'pointer struct' value in step 7

type of origin : one of int, selecs or joins,
where
int : originated by inter, diff or union views,
selecs : originated by selection views,

joins : originated by join views.

1.
id, indx, id.popt 2selecs
density 1.25313731025000e+07
profit 2.50627462050000e+07
space 2

2.
id, indx, id.popt 3selecs
density 3.14747792050000e+07
profit 1.89868484737095e+07
space 1

3.
id, indx, id.popt 1selecs
density 7.67441413000000e+06
profit 4.26706996121650e+06
space 1

indexes of the chosen access structures :

2
1

approximately optimal total profit : 4.40495946787095e+07

9. COSTS OF CANDIDATE PLACEMENT ACCESS STRUCTURES AFTER CONSIDERING THE EFFECTS OF THE VIEWS WHOSE REPRESENTATIVE ACCESS STRUCTURES ARE NOT CHOSEN IN STEP 8.

```
cluster :          1
  total cost :  2.95700070411180e+04
  number of views :          1
  start add. :          1

sorted :          1
  total cost :  1.19325212758587e+04
  number of views :          1
  start add. :          3

wellplaced :          1
wlpl structure no:          1
  total cost :  1.17469291907004e+05
  number of views :          1
  start add. :          2
  associated relations :          1,          3

cluster :          2
  total cost :  3.66250000000000e+02
  number of views :          1
  start add. :          4

wellplaced :          2
wlpl structure no:          1
  total cost :  6.92352818868020e+04
  number of views :          1
  start add. :          5
  associated relations :          2,          3
```

```
cluster :          3
  total cost :  6.84263279555968e+04
  number of views :          1
  start add. :          6

sorted :          3
  total cost :  2.22968572980426e+04
  number of views :          1
  start add. :          7

wellplaced :          3
wlpl structure no:          1
  total cost :  1.17469291907004e+05
  number of views :          1
  start add. :          2
  associated relations :          1,          3

wellplaced :          3
wlpl structure no:          2
  total cost :  6.92352818868020e+04
  number of views :          1
  start add. :          5
  associated relations :          2,          3
```

10. MINIMUM COST PLACEMENT ACCESS STRUCTURE CONFIGURATION OBTAINED USING THE COSTS FROM STEP 9.

```
rel : 1,   type : sorted

rel : 2,   type : cluster

rel : 3,   type : sorted

minimum total cost 3.51e+04
```

11. PROFITS OF POINTER ACCESS STRUCTURES OBTAINED USING THE PLACEMENT ACCESS STRUCTURE CONFIGURATION OF STEP 10.

```
pointer access structures in sorted order

  structure :          1
  density :  1.25313731025000e+07
  space :          2
  profit :  2.50627462050000e+07
  index to intpo, joinpo or selecpo :          2
  type of structure : selecs

  structure :          2
  density :  6.49891774241899e+06
  space :          1
  profit :  6.49891774241899e+06
  index to intpo, joinpo or selecpo :          3
  type of structure : selecs

  structure :          3
  density :  8.59725792433009e+05
  space :          1
  profit :  8.59725792433009e+05
  index to intpo, joinpo or selecpo :          1
  type of structure : selecs
```

12. THE RESULT OF THE POINTER ACCESS STRUCTURE OPTIMIZATION

```
space limit :          3

approximation tolerance :  1.00000000000000e-02
```

indexes of the pointer access structures chosen :

2
1

approximately optimal total profit :

3.15616639474190e+07

The optimal access structures are specified in steps 12 and 10. Specifically the access structures we obtain as a result of this design for the views given in step I.1 of the appendix are summarized in the following :

1. a sorted access structure on relation "Customers",
2. a cluster access structure on relation "Parts",
3. a sorted access structure on relation "Orders",
4. a pointer array access structure covering the whole range of the relation for processing selection views on attribute "part#" of relation "Parts",
5. a pointer array access structure just covering the range of the view 5 (in the input) for processing selection views on attribute "quantity" of relation "Orders".

The total profits of the pointer access structures is 3.15e+07. The total cost of placement access structures is 3.51e+04.

II. A LESS SIMPLE EXAMPLE OF METAMORPHOSIS AND THE OPTIMIZATION OF ACCESS STRUCTURES

Here we give a less simple example. This example has more views and relations in its input. Thus we think that it illustrates the metamorphosis and the optimization algorithms better then the previous one.

number of relations : 5

size of relations

rel :	1,	size :	20000
rel :	2,	size :	30000
rel :	3,	size :	10000
rel :	4,	size :	50000
rel :	5,	size :	5000

arity of relations

rel :	1,	number of attributes :	3
rel :	2,	number of attributes :	2
rel :	3,	number of attributes :	3
rel :	4,	number of attributes :	4
rel :	5,	number of attributes :	2

views :

v	rl1	rl2	at1	at2	size	pl	pi	pd	op
p	1				20000	3.0	4.0	3.0	
p	1				20000	1.0	1.0	5.0	
c	1	2			600000000	0.0	0.0	0.0	
c	1	3			200000000	0.0	1.0	1.0	
q	1	2			1000	1.0	2.0	1.0	
q	1	4			300	2.0	1.0	0.0	
q	1	5			800	2.0	0.0	0.0	
q	1	5			800	1.0	7.0	4.0	
j	1	2	1	2	5000	5000.0	7000.0	6000.0	EQ
j	1	3	2	3	1000	8050.0	10040.0	2550.0	EQ

j	1	5	2	2	1000	15050.0	25000.0	5350.0	EQ
p	2				30000	1.0	2.0	3.0	
c	2	3			300000000	0.0	1.0	0.0	
c	2	4			1500000000	1.0	2.0	0.0	
c	1	2			600000000	1.0	0.0	1.0	
q	2	3			100	1.0	2.0	0.0	
q	2	5			200	1.0	0.0	0.0	
q	1	2			1000	0.0	1.0	0.0	
j	2	3	1	2	1000	55500.0	76600.0	25600.0	EQ
j	2	4	2	4	2000	63300.0	44400.0	35500.0	EQ
j	2	5	2	2	3000	25550.0	35550.0	15660.0	EQ
p	3				10000	1.0	2.0	0.0	
c	3	4			500000000	1.0	0.0	0.0	
c	1	3			200000000	1.0	0.0	1.0	
c	2	3			300000000	0.0	1.0	0.0	
q	3	5			500	2.0	1.0	1.0	
q	2	3			100	1.0	2.0	3.0	
j	3	5	3	1	15000	50440.0	655500.0	74400.0	EQ
p	4				50000	1.0	5.0	9.0	
c	4	5			250000000	1.0	1.0	0.0	
c	2	4			1500000000	1.0	1.0	0.0	
c	3	4			500000000	1.0	1.0	1.0	
q	4	5			500	0.0	1.0	1.0	
q	1	4			300	2.0	1.0	0.0	
j	4	5	3	2	1000	34540.0	44450.0	45625.0	EQ
j	4	5	2	1	750	14565.0	44435.0	46785.0	EQ
p	5				5000	1.0	2.0	1.0	
c	4	5			250000000	1.0	1.0	0.0	
q	1	5			800	1.0	1.0	4.0	
q	2	5			200	1.0	1.0	1.0	
q	3	5			500	1.0	1.0	1.0	
q	4	5			500	1.0	1.0	1.0	
i	1	2			1500	553.0	560.0	750.0	
d	1	2			1500	350.0	440.0	100.0	
u	1	2			1500	200.0	300.0	400.0	
i	1	3			1000	350.0	600.0	50.0	
u	1	3			1000	150.0	250.0	100.0	
i	1	4			750	550.0	650.0	750.0	
d	1	4			750	350.0	400.0	500.0	
i	1	5			300	500.0	600.0	100.0	
i	2	3			500	600.0	450.0	250.0	

d	2	4			600	450.0	550.0	240.0	
i	2	5			800	450.0	450.0	450.0	
i	3	4			800	450.0	450.0	550.0	
i	3	5			780	750.0	789.0	650.0	
j	1	2	2	1	100000	500.0	400.0	300.0	GT
j	1	4	1	1	90000	400.0	350.0	250.0	LT
j	1	5	3	1	50000	700.0	600.0	300.0	LE
j	2	3	2	1	80000	250.0	500.0	150.0	GE
j	2	4	1	3	120000	220.0	330.0	450.0	LE
j	3	5	2	2	60000	173.0	150.0	20.0	LT
s	1		3		1000	500.0	354.0	250.0	LT
s	1		3		200	356.0	459.0	750.0	GT
s	3		3		1000	780.0	100.0	1000.0	GT
s	3		3		330	750.0	1200.0	29.0	GT
s	3		3		450	1000.0	1200.0	250.0	GE
s	3		3		500	1000.0	1200.0	250.0	GT
s	3		3		400	1000.0	1200.0	250.0	LT
s	3		3		300	1000.0	1200.0	250.0	LE
s	3		3		200	1000.0	1200.0	250.0	LE
s	3		3		230	1000.0	1200.0	250.0	LT
s	3		3		330	1000.0	1200.0	250.0	LE
s	3		3		440	1000.0	1200.0	250.0	LE
s	3		3		1200	1000.0	1200.0	250.0	LT
s	3		3		345	1000.0	1200.0	250.0	LE
s	3		3		234	1000.0	1200.0	250.0	LE
s	3		3		234	1000.0	1200.0	250.0	GE
s	3		3		567	1000.0	1200.0	250.0	LT
s	3		3		345	1000.0	1200.0	250.0	LE
s	3		3		345	1000.0	1200.0	250.0	LE
s	3		3		467	1000.0	1200.0	250.0	GE
s	3		3		789	1000.0	1200.0	250.0	LT
s	3		3		990	1000.0	1200.0	250.0	LE
s	3		3		1200	1000.0	1200.0	250.0	GT
s	3		3		679	1000.0	1200.0	250.0	GE
s	3		3		777	1000.0	1200.0	250.0	LE
s	3		3		998	1000.0	1200.0	250.0	LT
s	3		3		1569	1000.0	1200.0	250.0	GE
s	3		3		1559	1000.0	1200.0	250.0	GE
s	3		3		1559	1000.0	1200.0	250.0	LE

s	3	3	592	1000.0	1200.0	250.0	LE
s	3	3	591	1000.0	1200.0	250.0	LE
s	5	1	250	1200.0	300.0	49.0	GT
s	5	1	250	1200.0	300.0	49.0	LT
s	5	1	250	2200.0	3330.0	400.0	GT
s	5	1	250	2200.0	3330.0	400.0	LT
s	5	1	250	1200.0	300.0	49.0	GT
s	5	1	250	3456.0	7890.0	456.0	GE
s	5	1	250	1550.0	2900.0	350.0	LE
s	5	1	250	2450.0	3450.0	670.0	GT
s	5	1	250	1200.0	300.0	49.0	GE
s	5	1	250	2450.0	3450.0	670.0	GT
s	5	1	250	1550.0	2900.0	350.0	LT
s	5	1	250	1200.0	300.0	49.0	GT
s	5	1	250	2200.0	3330.0	400.0	GE
s	5	1	250	2450.0	3450.0	670.0	LE
s	5	1	250	1550.0	2900.0	350.0	GT
s	5	1	250	1200.0	300.0	49.0	LT
s	5	1	250	1550.0	2900.0	350.0	LE
s	5	1	250	2450.0	3450.0	670.0	GE
s	5	1	250	2200.0	3330.0	400.0	GT
s	5	1	250	1200.0	300.0	49.0	LT
s	5	1	250	1550.0	2900.0	350.0	GT
s	5	1	250	1550.0	2900.0	350.0	GT
s	5	1	255	2200.0	3330.0	400.0	GE

2. CANDIDATE ACCESS STRUCTURES GENERATED BY THE METAMORPHOSIS

placement access structures

The value of "number of views" is the number of constituent views of the access structure.

A well-placed access structure is shown at both of its associated relations.

"poarray" stands for pointer array access structure.

```
relation :          1

  cluster
    total cost :  8.17290862496725e+08
    number of views :          4

  sorted
    total cost :  1.27112162854860e+08
    number of views :          4


relation :          2

  cluster
    total cost :  2.92922165962074e+09
    number of views :          4

  sorted
    total cost :  6.20006986197293e+08
    number of views :          3


relation :          3

  cluster
    total cost :  1.03411269459327e+09
    number of views :          4

  sorted
    total cost :  2.81033076395359e+09
    number of views :          2
```

```
relation :          4

  cluster
    total cost :  9.63785540001154e+09
    number of views :          4

  sorted
    total cost :  2.48491650149846e+09
    number of views :          2


relation :          5

  cluster
    total cost :  2.01405539121939e+09
    number of views :          2

  sorted
    total cost :  1.29175013033308e+09
    number of views :          4


relation : 1

  well-placed  no. :          1
    total cost :  1.21783297595168e+09
    number of views :          1
    relations :          1,         2

  well-placed  no. :          2
    total cost :  5.31125318547905e+08
    number of views :          1
    relations :          1,         3

  well-placed  no. :          3
    total cost :  2.20465613825351e+08
    number of views :          1
    relations :          1,         5


relation : 2

  well-placed  no. :          1
    total cost :  1.24806750675828e+09
    number of views :          1
    relations :          2,         3

  well-placed  no. :          2
    total cost :  5.96285018987445e+08
    number of views :          1
    relations :          2,         4
```

```
  well-placed  no. :          3
    total cost :  9.40980956898504e+08
    number of views :          1
    relations :          2,         5

  well-placed  no. :          4
    total cost :  1.21783297595168e+09
    number of views :          1
    relations :          1,         2

relation : 3

  well-placed  no. :          1
    total cost :  1.16263300408992e+10
    number of views :          1
    relations :          3,         5

  well-placed  no. :          2
    total cost :  5.31125318547905e+08
    number of views :          1
    relations :          1,         3

  well-placed  no. :          3
    total cost :  1.24806750675828e+09
    number of views :          1
    relations :          2,         3


relation : 4

  well-placed  no. :          1
    total cost :  1.48821651648317e+08
    number of views :          2
    relations :          4,         5

  well-placed  no. :          2
    total cost :  5.96285018987445e+08
    number of views :          1
    relations :          2,         4


relation : 5

  well-placed  no. :          1
    total cost :  2.20465613825351e+08
    number of views :          1
    relations :          1,         5
```

```
well-placed  no. :          2
  total cost :  9.40980956898504e+08
  number of views :          1
  relations :          2,          5

well-placed  no. :          3
  total cost :  1.16263300408992e+10
  number of views :          1
  relations :          3,          5

well-placed  no. :          4
  total cost :  1.48821651648317e+08
  number of views :          2
  relations :          4,          5
```

pointer access structures

pointer array access structures generated by intersection, difference and union views

```
acc. structure no. :          1
  cost :  2.74056600000000e+04
  storage :          8
  rel1 :          1
  rel2 :          2
  number of views :          3

acc. structure no. :          2
  cost :  7.50612000000000e+03
  storage :          6
  rel1 :          1
  rel2 :          3
  number of views :          2

acc. structure no. :          3
  cost :  1.20040800000000e+04
  storage :          4
  rel1 :          1
  rel2 :          4
  number of views :          2

acc. structure no. :          4
  cost :  1.80204000000000e+03
  storage :          2
  rel1 :          1
  rel2 :          5
  number of views :          1
```

```
acc. structure no. :          5
  cost :  3.25306000000000e+03
  storage :          3
  rel1 :          2
  rel2 :          3
  number of views :          1

acc. structure no. :          6
  cost :  3.72408000000000e+03
  storage :          4
  rel1 :          2
  rel2 :          4
  number of views :          1

acc. structure no. :          7
  cost :  5.40510000000000e+03
  storage :          5
  rel1 :          2
  rel2 :          5
  number of views :          1

acc. structure no. :          8
  cost :  5.80510000000000e+03
  storage :          5
  rel1 :          3
  rel2 :          4
  number of views :          1

acc. structure no. :          9
  cost :  8.54118000000000e+03
  storage :          4
  rel1 :          3
  rel2 :          5
  number of views :          1


access structures generated by join views

acc. structure no. :          1
  poarray structure
    cost :  2.56964610115858e+09
    storage :        501
    relation 1 :          1
    relation 2 :          2
    number of views :          1
    attribute 1 :          2
    attribute 2 :          1
```

```
circular chain
  cost :  2.5696461011585 8e+09
  storage :        501
  relation 1 :          1
  relation 2 :          2
  number of views :          1
  attribute 1 :          2
  attribute 2 :          1

acc. structure no. :          2
  poarray structure
    cost :  1.40559295487071e+09
    storage :        451
    relation 1 :          1
    relation 2 :          4
    number of views :          1
    attribute 1 :          1
    attribute 2 :          1

circular chain
  cost :  1.40559295487071e+09
  storage :        451
  relation 1 :          1
  relation 2 :          4
  number of views :          1
  attribute 1 :          1
  attribute 2 :          1

acc. structure no. :          3
  poarray structure
    cost :  2.11582135511380e+10
    storage :        251
    relation 1 :          1
    relation 2 :          5
    number of views :          1
    attribute 1 :          3
    attribute 2 :          1

circular chain
  cost :  2.11582135511380e+10
  storage :        251
  relation 1 :          1
  relation 2 :          5
  number of views :          1
  attribute 1 :          3
  attribute 2 :          1
```

```
acc. structure no. :          4
  poarray structure
    cost :  2.09285894479717e+10
    storage :        401
    relation 1 :          2
    relation 2 :          3
    number of views :          1
    attribute 1 :          2
    attribute 2 :          1

circular chain
  cost :  2.09285894479717e+10
  storage :        401
  relation 1 :          2
  relation 2 :          3
  number of views :          1
  attribute 1 :          2
  attribute 2 :          1

acc. structure no. :          5
  poarray structure
    cost :  4.03622314409020e+09
    storage :        601
    relation 1 :          2
    relation 2 :          4
    number of views :          1
    attribute 1 :          1
    attribute 2 :          3

circular chain
  cost :  4.03622314409020e+09
  storage :        601
  relation 1 :          2
  relation 2 :          4
  number of views :          1
  attribute 1 :          1
  attribute 2 :          3

acc. structure no. :          6
  poarray structure
    cost :  2.27605209719223e+10
    storage :        301
    relation 1 :          3
    relation 2 :          5
    number of views :          1
    attribute 1 :          2
    attribute 2 :          2
```

```
circular chain
  cost :  2.27605209719223e+10
  storage :        301
  relation 1 :          3
  relation 2 :          5
  number of views :          1
  attribute 1 :          2
  attribute 2 :          2
```

access structure sets generated for attributes by selection views:

For each attribute there are eight separate access structure sets as explained on page 58f.

In the following the information given under "associated views" is the information about the constituent views. More specifically, "view address" gives the location of the view in the input view set and "strc. no." shows in which compound access structure in the attribute access structure set the view takes part. For example, "strc. no. 2" in an array tree access structure set means that the view is a constituent of the second access structure in the attribute access structure set (i.e. tree).

```
rel :          1    attribute :          3

  tree
    cost :  5.99510314215626e+05
    storage :        101
    relation 1 :          1
    number of views :          2
    attribute 1 :          3

  poarray
    cost :  6.51181000000000e+05
    storage :        101
    relation 1 :          1
    number of views :          2
    attribute 1 :          3

  chain
    cost :  5.75976000000000e+03
    storage :        101
    relation 1 :          1
    number of views :          2
    attribute 1 :          3
```

```
aggre. of poarray
  cost :  7.093160000000000e+03
  storage :          8
  relation 1 :          1
  number of views :          2
  attribute 1 :          3

array and tree
  cost :  9.83791314215626e+05
  storage :        202
  relation 1 :          1
  number of views          2
  attribute 1 :          3
  associated views :

    view address   62,   strc. no    2
    view address   63,   strc. no    2

array and chain
  cost :  3.90040760000000e+05
  storage :        202
  relation 1 :          1
  number of views          2
  attribute 1 :          3
  associated views :

    view address   62,   strc. no    2
    view address   63,   strc. no    2

aggr. of arrays and tree
  cost :  5.99510314215626e+05
  storage :        101
  relation 1 :          1
  number of views          2
  attribute 1 :          3
  associated views :

    view address   62,   strc. no    2
    view address   63,   strc. no    2

aggr. of arrays and chain
  cost :  1.10399200000000e+04
  storage :        109
  relation 1 :          1
  number of views          2
  attribute 1 :          3
  associated views :

    view address   62,   strc. no    1
    view address   63,   strc. no    1
```

```
rel :          3    attribute :          3

  tree
    cost :  5.7798154691508 3e+06
    storage :          51
    relation 1 :           3
    number of views :          29
    attribute 1 :           3

  poarray
    cost :  6.6166670000000e+06
    storage :          51
    relation 1 :           3
    number of views :          29
    attribute 1 :           3

  chain
    cost :  1.0385528000000e+05
    storage :          51
    relation 1 :           3
    number of views :          29
    attribute 1 :           3

  aggre. of poarray
    cost :  2.3193005000000e+05
    storage :         110
    relation 1 :           3
    number of views :          29
    attribute 1 :           3

  array and tree
    cost :  8.8960324691508 3e+06
    storage :         102
    relation 1 :           3
    number of views          29
    attribute 1 :           3
    associated views :

      view address   64,   strc. no    2
      view address   65,   strc. no    2
      view address   66,   strc. no    2
      view address   67,   strc. no    2
      view address   68,   strc. no    2
      view address   69,   strc. no    2
      view address   70,   strc. no    2
      view address   71,   strc. no    2
      view address   72,   strc. no    2
      view address   73,   strc. no    2
      view address   74,   strc. no    2
      view address   75,   strc. no    2
```

```
    view address   76,   strc. no    2
    view address   77,   strc. no    2
    view address   78,   strc. no    2
    view address   79,   strc. no    2
    view address   80,   strc. no    2
    view address   81,   strc. no    2
    view address   82,   strc. no    2
    view address   83,   strc. no    2
    view address   84,   strc. no    2
    view address   85,   strc. no    2
    view address   86,   strc. no    2
    view address   87,   strc. no    2
    view address   88,   strc. no    2
    view address   89,   strc. no    2
    view address   90,   strc. no    2
    view address   91,   strc. no    2
    view address   92,   strc. no    2

array and chain
  cost :  3.22007228000000e+06
  storage :        102
  relation 1 :          3
  number of views         29
  attribute 1 :          3
  associated views :

    view address   64,   strc. no    2
    view address   65,   strc. no    2
    view address   66,   strc. no    2
    view address   67,   strc. no    2
    view address   68,   strc. no    2
    view address   69,   strc. no    2
    view address   70,   strc. no    2
    view address   71,   strc. no    2
    view address   72,   strc. no    2
    view address   73,   strc. no    2
    view address   74,   strc. no    2
    view address   75,   strc. no    2
    view address   76,   strc. no    2
    view address   77,   strc. no    2
    view address   78,   strc. no    2
    view address   79,   strc. no    2
    view address   80,   strc. no    2
    view address   81,   strc. no    2
    view address   82,   strc. no    2
    view address   83,   strc. no    2
    view address   84,   strc. no    2
    view address   85,   strc. no    2
    view address   86,   strc. no    2
```

```
    view address   87,   strc. no    2
    view address   88,   strc. no    2
    view address   89,   strc. no    2
    view address   90,   strc. no    2
    view address   91,   strc. no    2
    view address   92,   strc. no    2

aggr. of arrays and tree
  cost :  5.7798154691508 3e+06
  storage :          51
  relation 1 :          3
  number of views         29
  attribute 1 :          3
  associated views :

    view address   64,   strc. no    2
    view address   65,   strc. no    2
    view address   66,   strc. no    2
    view address   67,   strc. no    2
    view address   68,   strc. no    2
    view address   69,   strc. no    2
    view address   70,   strc. no    2
    view address   71,   strc. no    2
    view address   72,   strc. no    2
    view address   73,   strc. no    2
    view address   74,   strc. no    2
    view address   75,   strc. no    2
    view address   76,   strc. no    2
    view address   77,   strc. no    2
    view address   78,   strc. no    2
    view address   79,   strc. no    2
    view address   80,   strc. no    2
    view address   81,   strc. no    2
    view address   82,   strc. no    2
    view address   83,   strc. no    2
    view address   84,   strc. no    2
    view address   85,   strc. no    2
    view address   86,   strc. no    2
    view address   87,   strc. no    2
    view address   88,   strc. no    2
    view address   89,   strc. no    2
    view address   90,   strc. no    2
    view address   91,   strc. no    2
    view address   92,   strc. no    2
```

```
aggr. of arrays and chain
  cost :  2.46646200000000e+05
  storage :        122
  relation 1 :          3
  number of views         29
  attribute 1 :          3
  associated views :

    view address   64,   strc. no    2
    view address   65,   strc. no    2
    view address   66,   strc. no    2
    view address   67,   strc. no    2
    view address   68,   strc. no    2
    view address   69,   strc. no    2
    view address   70,   strc. no    2
    view address   71,   strc. no    2
    view address   72,   strc. no    2
    view address   73,   strc. no    2
    view address   74,   strc. no    1
    view address   75,   strc. no    1
    view address   76,   strc. no    2
    view address   77,   strc. no    2
    view address   78,   strc. no    1
    view address   79,   strc. no    1
    view address   80,   strc. no    1
    view address   81,   strc. no    2
    view address   82,   strc. no    1
    view address   83,   strc. no    1
    view address   84,   strc. no    1
    view address   85,   strc. no    2
    view address   86,   strc. no    1
    view address   87,   strc. no    1
    view address   88,   strc. no    1
    view address   89,   strc. no    1
    view address   90,   strc. no    1
    view address   91,   strc. no    1
    view address   92,   strc. no    1


rel :          5    attribute :          1

  tree
    cost :  4.02807166139330e+06
    storage :         26
    relation 1 :          5
    number of views :         23
    attribute 1 :          1
```

```
poarray
  cost :  4.41516400000000e+06
  storage :          26
  relation 1 :           5
  number of views :          23
  attribute 1 :           1

chain
  cost :  1.34676480000000e+05
  storage :          26
  relation 1 :           5
  number of views :          23
  attribute 1 :           1

aggre. of poarray
  cost :  1.34413920000000e+05
  storage :          46
  relation 1 :           5
  number of views :          23
  attribute 1 :           1

array and tree
  cost :  5.75886066139330e+06
  storage :          52
  relation 1 :           5
  number of views          23
  attribute 1 :           1
  associated views :

    view address   93,   strc. no    2
    view address   94,   strc. no    2
    view address   95,   strc. no    2
    view address   96,   strc. no    2
    view address   97,   strc. no    2
    view address   98,   strc. no    2
    view address   99,   strc. no    2
    view address  100,   strc. no    2
    view address  101,   strc. no    2
    view address  102,   strc. no    2
    view address  103,   strc. no    2
    view address  104,   strc. no    2
    view address  105,   strc. no    2
    view address  106,   strc. no    2
    view address  107,   strc. no    2
    view address  108,   strc. no    2
    view address  109,   strc. no    2
    view address  110,   strc. no    2
    view address  111,   strc. no    2
```

```
    view address  112,   strc. no    2
    view address  113,   strc. no    2
    view address  114,   strc. no    2
    view address  115,   strc. no    2

array and chain
  cost :  1.865465480000000e+06
  storage :          52
  relation 1 :           5
  number of views          23
  attribute 1 :           1
  associated views :

    view address   93,   strc. no    2
    view address   94,   strc. no    2
    view address   95,   strc. no    2
    view address   96,   strc. no    2
    view address   97,   strc. no    2
    view address   98,   strc. no    2
    view address   99,   strc. no    2
    view address  100,   strc. no    2
    view address  101,   strc. no    2
    view address  102,   strc. no    2
    view address  103,   strc. no    2
    view address  104,   strc. no    2
    view address  105,   strc. no    2
    view address  106,   strc. no    2
    view address  107,   strc. no    2
    view address  108,   strc. no    2
    view address  109,   strc. no    2
    view address  110,   strc. no    2
    view address  111,   strc. no    2
    view address  112,   strc. no    2
    view address  113,   strc. no    2
    view address  114,   strc. no    2
    view address  115,   strc. no    2

aggr. of arrays and tree
  cost :  4.02807166139330e+06
  storage :          26
  relation 1 :           5
  number of views          23
  attribute 1 :           1
  associated views :

    view address   93,   strc. no    2
    view address   94,   strc. no    2
    view address   95,   strc. no    2
    view address   96,   strc. no    2
```

```
    view address   97,   strc. no    2
    view address   98,   strc. no    2
    view address   99,   strc. no    2
    view address  100,   strc. no    2
    view address  101,   strc. no    2
    view address  102,   strc. no    2
    view address  103,   strc. no    2
    view address  104,   strc. no    2
    view address  105,   strc. no    2
    view address  106,   strc. no    2
    view address  107,   strc. no    2
    view address  108,   strc. no    2
    view address  109,   strc. no    2
    view address  110,   strc. no    2
    view address  111,   strc. no    2
    view address  112,   strc. no    2
    view address  113,   strc. no    2
    view address  114,   strc. no    2
    view address  115,   strc. no    2

aggr. of arrays and chain
  cost :  1.61295970000000e+05
  storage :          38
  relation 1 :           5
  number of views          23
  attribute 1 :           1
  associated views :

    view address   93,   strc. no    2
    view address   94,   strc. no    2
    view address   95,   strc. no    2
    view address   96,   strc. no    2
    view address   97,   strc. no    2
    view address   98,   strc. no    1
    view address   99,   strc. no    2
    view address  100,   strc. no    1
    view address  101,   strc. no    2
    view address  102,   strc. no    1
    view address  103,   strc. no    2
    view address  104,   strc. no    2
    view address  105,   strc. no    2
    view address  106,   strc. no    1
    view address  107,   strc. no    2
    view address  108,   strc. no    2
    view address  109,   strc. no    2
    view address  110,   strc. no    1
    view address  111,   strc. no    2
    view address  112,   strc. no    2
```

view address	113,	strc. no	2
view address	114,	strc. no	2
view address	115,	strc. no	1

3. MINIMUM COST SORTED OR PLACEMENT TYPE ACCESS STRUCTURE CONFIGURATION

rel : 1, type : sorted

rel : 2, type : sorted

rel : 3, type : cluster

rel : 4, type : sorted

rel : 5, type : sorted

4. PROFITS OF THE POINTER ACCESS STRUCTURES FOR THE PLACEMENT CONFIGURATION OF STEP 3

abbreviations :

intpo : data structure storing pointer access structures generated by inter, diff and union views.

joinpo : data structure storing pointer access structures generated by join views.

selecpo : data structure storing pointer access structures generated by selection views.

The last two entries for each access structure give us the location of the access structure in intpo, joinpo or selecpo. For example,

index to intpo, joinpo or selecpo : 3
type of structure : int

implies that the access structure is the third element of intpo.

pointer access structures in sorted order

```
structure :          1
density :  7.41293928648000e+09
space :          4
profit :  2.96517571459200e+10
index to intpo, joinpo or selecpo :          3
type of structure : int

structure :          2
density :  3.98971992770808e+09
space :          5
profit :  1.99485996385404e+10
index to intpo, joinpo or selecpo :          8
type of structure : int

structure :          3
density :  2.91927480840045e+09
space :          4
profit :  1.16770992336018e+10
index to intpo, joinpo or selecpo :          9
type of structure : int

structure :          4
density :  1.85249976126571e+09
space :          3
profit :  5.55749928379713e+09
index to intpo, joinpo or selecpo :          5
type of structure : int

structure :          5
density :  1.47136483898000e+09
space :          4
profit :  5.88545935592000e+09
index to intpo, joinpo or selecpo :          6
type of structure : int

structure :          6
density :  1.44189921804250e+09
space :          8
profit :  1.15351937443400e+10
index to intpo, joinpo or selecpo :          1
type of structure : int
```

```
structure :          7
density :  5.68570535135509e+08
space :          6
profit :  3.41142321081305e+09
index to intpo, joinpo or selecpo :          2
type of structure : int

structure :          8
density :  4.84276258980000e+08
space :          5
profit :  2.42138129490000e+09
index to intpo, joinpo or selecpo :          7
type of structure : int

structure :          9
density :  4.51316901256487e+08
space :         51
profit :  2.30171619640808e+10
index to intpo, joinpo or selecpo :          2
type of structure : selecs
alternate no. :          7

structure :         10
density :  3.00231598980000e+08
space :          2
profit :  6.00463197960000e+08
index to intpo, joinpo or selecpo :          4
type of structure : int

structure :         11
density :  4.81498002348087e+07
space :         26
profit :  1.25189480610503e+09
index to intpo, joinpo or selecpo :          3
type of structure : selecs
alternate no. :          3

structure :         12
density :  2.59566518557682e+07
space :          8
profit :  2.07653214846146e+08
index to intpo, joinpo or selecpo :          1
type of structure : selecs
alternate no. :          4
```

```
structure :          13
density :  9.72149848960865e+06
space :         451
profit :  4.38439581881350e+09
index to intpo, joinpo or selecpo :          2
type of structure : joins
alternate no. :          2

structure :          14
density :  9.60866319137282e+06
space :         601
profit :  5.77480657801506e+09
index to intpo, joinpo or selecpo :          5
type of structure : joins
alternate no. :          2

structure :          15
density : -5.88293944953569e+05
space :         501
profit : -2.94735266421738e+08
index to intpo, joinpo or selecpo :          1
type of structure : joins
alternate no. :          2

structure :          16
density : -5.24814609920569e+07
space :         401
profit : -2.10450658578148e+10
index to intpo, joinpo or selecpo :          4
type of structure : joins
alternate no. :          2

structure :          17
density : -9.22263110043638e+07
space :         301
profit : -2.77601196123135e+10
index to intpo, joinpo or selecpo :          6
type of structure : joins
alternate no. :          2

structure :          18
density : -9.64031928059598e+07
space :         251
profit : -2.41972013942959e+10
index to intpo, joinpo or selecpo :          3
type of structure : joins
alternate no. :          2
```

5. MINIMUM COST PLACEMENT ACCESS STRUCTURE CONFIGURATION WITHOUT CONSIDERING EXTRA VIEWS

```
rel : 1,   type : well-placed with rel. 3
rel : 2,   type : sorted
rel : 3,   type : well-placed with rel. 1
rel : 4,   type : well-placed with rel. 5
rel : 5,   type : well-placed with rel. 4
```

6. PROFITS OF POINTER ACCESS STRUCTURES USING THE PLACEMENT CONFIGURATION OF STEP 5.

```
pointer access structures in sorted order

  structure :          1
  density :  8.42962182398000e+09
  space :          4
  profit :  3.37184872959200e+10
  index to intpo, joinpo or selecpo :          3
  type of structure : int

  structure :          2
  density :  6.67948601470351e+09
  space :         26
  profit :  1.73666636382291e+11
  index to intpo, joinpo or selecpo :          3
  type of structure : selecs
  alternate no. :          7
```

```
structure :          3
density :  2.97017174564667e+09
space :          5
profit :  1.48508587282333e+10
index to intpo, joinpo or selecpo :          8
type of structure : int
alternate no. :          0

structure :          4
density :  2.94460612570500e+09
space :          4
profit :  1.17784245028200e+10
index to intpo, joinpo or selecpo :          9
type of structure : int

structure :          5
density :  1.89249838991750e+09
space :          8
profit :  1.51399871193400e+10
index to intpo, joinpo or selecpo :          1
type of structure : int

structure :          6
density :  1.49553981398000e+09
space :          4
profit :  5.98215925592000e+09
index to intpo, joinpo or selecpo :          6
type of structure : int

structure :          7
density :  1.36105931898000e+09
space :          5
profit :  6.80529659490000e+09
index to intpo, joinpo or selecpo :          7
type of structure : int

structure :          8
density :  1.26458540502904e+09
space :         51
profit :  6.44938556564808e+10
index to intpo, joinpo or selecpo :          2
type of structure : selecs
alternate no. :          7

structure :          9
density :  1.03506689898000e+09
space :          2
profit :  2.07013379796000e+09
index to intpo, joinpo or selecpo :          4
type of structure : int
```

```
structure :          10
density :  9.45499170505000e+08
space :          8
profit :  7.56399336404000e+09
index to intpo, joinpo or selecpo :          1
type of structure : selecs
alternate no. :          4

structure :          11
density :  8.45550985091111e+08
space :          3
profit :  2.53665295527333e+09
index to intpo, joinpo or selecpo :          5
type of structure : int

structure :          12
density :  3.50305123980000e+08
space :          6
profit :  2.10183074388000e+09
index to intpo, joinpo or selecpo :          2
type of structure : int

structure :          13
density :  1.13414796659419e+07
space :        451
profit :  5.11500732933982e+09
index to intpo, joinpo or selecpo :          2
type of structure : joins
alternate no. :          2

structure :          14
density :  9.89369666715879e+06
space :        601
profit :  5.94611169696243e+09
index to intpo, joinpo or selecpo :          5
type of structure : joins
alternate no. :          2

structure :          15
density :  7.17624964595753e+05
space :        501
profit :  3.59530107262472e+08
index to intpo, joinpo or selecpo :          1
type of structure : joins
alternate no. :          2
```

```
structure :          16
density : -5.72693506118206e+07
space :         401
profit : -2.29650095953401e+10
index to intpo, joinpo or selecpo :          4
type of structure : joins
alternate no. :          2

structure :          17
density : -9.11795660498264e+07
space :         251
profit : -2.28860710785064e+10
index to intpo, joinpo or selecpo :          3
type of structure : joins
alternate no. :          2

structure :          18
density : -1.16200032819815e+08
space :         301
profit : -3.49762098787644e+10
index to intpo, joinpo or selecpo :          6
type of structure : joins
alternate no. :          2
```

7. AVERAGE OF THE PROFITS FROM STEPS 6 AND 4

Pointer access structures are listed in the order they appear in steps 4 and 6.

```
pointer  struct          1
profit from step 6 :   3.37184872959200e+10
profit from step 4 :   2.96517571459200e+10
average profit :       3.16851222209200e+10

pointer  struct          2
profit from step 6 :   1.73666636382291e+11
profit from step 4 :   1.99485996385404e+10
average profit :       9.68076180104158e+10

pointer  struct          3
profit from step 6 :   1.48508587282333e+10
profit from step 4 :   1.16770992336018e+10
average profit :       1.32639789809176e+10
```

```
pointer  struct          4
profit from step 6 :   1.17784245028200e+10
profit from step 4 :   5.55749928379713e+09
average profit :       8.66796189330856e+09

pointer  struct          5
profit from step 6 :   1.51399871193400e+10
profit from step 4 :   5.88545935592000e+09
average profit :       1.05127232376300e+10

pointer  struct          6
profit from step 6 :   5.98215925592000e+09
profit from step 4 :   1.15351937443400e+10
average profit :       8.75867650013000e+09

pointer  struct          7
profit from step 6 :   6.80529659490000e+09
profit from step 4 :   3.41142321081305e+09
average profit :       5.10835990285653e+09

pointer  struct          8
profit from step 6 :   6.44938556564808e+10
profit from step 4 :   2.42138129490000e+09
average profit :       3.34576184756904e+10

pointer  struct          9
profit from step 6 :   2.07013379796000e+09
profit from step 4 :   2.30171619640808e+10
average profit :       1.25436478810204e+10

pointer  struct         10
profit from step 6 :   7.56399336404000e+09
profit from step 4 :   6.00463197960000e+08
average profit :       4.08222828100000e+09

pointer  struct         11
profit from step 6 :   2.53665295527333e+09
profit from step 4 :   1.25189480610503e+09
average profit :       1.89427388068918e+09

pointer  struct         12
profit from step 6 :   2.10183074388000e+09
profit from step 4 :   2.07653214846146e+08
average profit :       1.15474197936307e+09

pointer  struct         13
profit from step 6 :   5.11500732933982e+09
profit from step 4 :   4.38439581881350e+09
average profit :       4.74970157407666e+09
```

```
pointer  struct          14
profit from step 6 :     5.94611169696243e+09
profit from step 4 :     5.77480657801506e+09
average profit :         5.86045913748875e+09

pointer  struct          15
profit from step 6 :     3.59530107262472e+08
profit from step 4 :    -2.94735266421738e+08
average profit :         3.23974204203672e+07

pointer  struct          16
profit from step 6 :    -2.29650095953401e+10
profit from step 4 :    -2.10450658578148e+10
average profit :        -2.20050377265774e+10

pointer  struct          17
profit from step 6 :    -2.28860710785064e+10
profit from step 4 :    -2.77601196123135e+10
average profit :        -2.53230953454100e+10

pointer  struct          18
profit from step 6 :    -3.49762098787644e+10
profit from step 4 :    -2.41972013942959e+10
average profit :        -2.95867056365302e+10
```

8. THE RESULT OF THE POINTER ACCESS STRUCTURE OPTIMIZATION

```
space limit :          20

approximation tolerance :  5.00000000000000e-02

number of pointer access structures :        18
```

abbreviations :

id : 'pointer struct' value in step 7

type of origin : one of int, selecs or joins,
where
int : originated by inter, diff or union views,
selecs : originated by selection views,

joins : originated by join views.

1.
id, type of origin : 3, int
density : 8.429621823980000e+09
profit : 3.16851222209200e+10
space : 4

2.
id, type of origin : 3, selecs
density : 6.679486014703351e+09
profit : 9.68076180104158e+10
space : 26

3.
id, type of origin : 8, int
density : 2.97017174564667e+09
profit : 1.32639789809176e+10
space : 5

4.
id, type of origin : 9, int
density : 2.94460612570500e+09
profit : 8.66796189330856e+09
space : 4

5.
id, type of origin : 1, int
density : 1.89249838991750e+09
profit : 1.05127232376300e+10
space : 8

6.
id, type of origin : 6, int
density : 1.49553981398000e+09
profit : 8.75867650013000e+09
space : 4

```
7.
id, type of origin :      7,  int
density :  1.36105931898000e+09
profit :   5.10835990285653e+09
space :         5

8.
id, type of origin :      2,  selecs
density :  1.26458540502904e+09
profit :   3.34576184756904e+10
space :        51

9.
id, type of origin :      4,  int
density :  1.03506689898000e+09
profit :   1.25436478810204e+10
space :         2

10.
id, type of origin :      1,  selecs
density :  9.45499170505000e+08
profit :   4.08222828100000e+09
space :         8

11.
id, type of origin :      5,  int
density :  8.45550985091111e+08
profit :   1.89427388068918e+09
space :         3

12.
id, type of origin :      2,  int
density :  3.50305123980000e+08
profit :   1.15474197936307e+09
space :         6

13.
id, type of origin :      2,  joins
density :  1.13414796659419e+07
profit :   4.74970157407666e+09
space :       451

14.
id, type of origin :      5,  joins
density :  9.89369666715879e+06
profit :   5.86045913748875e+09
space :       601
```

```
15.
id, type of origin :      1,  joins
density :  7.17624964595753e+05
profit :   3.23974204203672e+07
space :       501

16.
id, type of origin :      4,  joins
density : -5.72693506118206e+07
profit :  -2.20050377265774e+10
space :       401

17.
id, type of origin :      3,  joins
density : -9.11795660498264e+07
profit :  -2.53230953454100e+10
space :       251

18.
id, type of origin :      6,  joins
density : -1.16200032819815e+08
profit :  -2.95867056365302e+10
space :       301


id's of the chosen access structures :

         9
         6
         4
         3
         1


approximately optimal total profit :

    7.49193874762966e+10
```

9. COSTS OF CANDIDATE PLACEMENT ACCESS STRUCTURES AFTER CONSIDERING THE EFFECTS OF THE VIEWS WHOSE REPRESENTATIVE ACCESS STRUCTURES ARE NOT CHOSEN IN STEP 8.

```
cluster :          1
  total cost :  9.61987650257941e+08
  number of views :          4

sorted :          1
  total cost :  1.91425349358218e+08
  number of views :          4

well-placed :          1
wlpl structure no:          1
  total cost :  1.43279127219528e+09
  number of views :          1
  associated relations :          1,          2

well-placed :          1
wlpl structure no:          2
  total cost :  1.90961015700978e+09
  number of views :          1
  associated relations :          1,          3

well-placed :          1
wlpl structure no:          3
  total cost :  7.66816060700074e+08
  number of views :          1
  associated relations :          1,          5

cluster :          2
  total cost :  3.07588144972168e+09
  number of views :          4

sorted :          2
  total cost :  6.72232571089622e+08
  number of views :          3
```

```
well-placed :          2
wlpl structure no:          1
  total cost :  2.20935177123201e+09
  number of views :          1
  associated relations :          2,          3

well-placed :          2
wlpl structure no:          2
  total cost :  7.25842824532155e+08
  number of views :          1
  associated relations :          2,          4

well-placed :          2
wlpl structure no:          3
  total cost :  1.38955899114146e+09
  number of views :          1
  associated relations :          2,          5

well-placed :          2
wlpl structure no:          4
  total cost :  1.43279127219528e+09
  number of views :          1
  associated relations :          1,          2

cluster :          3
  total cost :  1.14843732999544e+09
  number of views :          4

sorted :          3
  total cost :  2.89098911929908e+09
  number of views :          2

well-placed :          3
wlpl structure no:          1
  total cost :  1.24646942654001e+10
  number of views :          1
  associated relations :          3,          5

well-placed :          3
wlpl structure no:          2
  total cost :  1.90961015700978e+09
  number of views :          1
  associated relations :          1,          3
```

```
well-placed :          3
wlpl structure no:          3
  total cost :  2.2093517712320 1e+09
  number of views :          1
  associated relations :          2,          3

cluster :          4
  total cost :  9.80491611849854e+09
  number of views :          4

sorted :          4
  total cost :  2.57412228315277e+09
  number of views :          2

well-placed :          4
wlpl structure no:          1
  total cost :  1.82026470207064e+09
  number of views :          2
  associated relations :          4,          5

well-placed :          4
wlpl structure no:          2
  total cost :  7.25842824532155e+08
  number of views :          1
  associated relations :          2,          4

cluster :          5
  total cost :  2.05168755543343e+09
  number of views :          2

sorted :          5
  total cost :  1.32169747956371e+09
  number of views :          4

well-placed :          5
wlpl structure no:          1
  total cost :  7.66816060700074e+08
  number of views :          1
  associated relations :          1,          5

well-placed :          5
wlpl structure no:          2
  total cost :  1.38955899114146e+09
  number of views :          1
  associated relations :          2,          5
```

```
well-placed :          5
wlpl structure no:          3
  total cost :  1.246469426540001e+10
  number of views :          1
  associated relations :          3,          5

well-placed :          5
wlpl structure no:          4
  total cost :  1.8202647020706 4e+09
  number of views :          2
  associated relations :          4,          5
```

10. MINIMUM COST PLACEMENT ACCESS STRUCTURE CONFIGURATION OBTAINED USING THE COSTS FROM STEP 9.

```
rel : 1,   type : well-placed with rel. 5

rel : 2,   type : well-placed with rel. 4

rel : 3,   type : cluster

rel : 4,   type : well-placed with rel. 2

rel : 5,   type : well-placed with rel. 1

minimum total cost : 2.639e+09
```

11. PROFITS OF POINTER ACCESS STRUCTURES OBTAINED USING THE PLACEMENT ACCESS STRUCTURE CONFIGURATION OF STEP 10.

```
pointer access structures in sorted order

  structure :          1
  density :  8.37515939898000e+09
  space :          4
  profit :  3.35006375959200e+10
  index to intpo, joinpo or selecpo :          3
  type of structure : int

  structure :          2
  density :  4.24524792770808e+09
  space :          5
  profit :  2.12262396385404e+10
  index to intpo, joinpo or selecpo :          8
  type of structure : int

  structure :          3
  density :  3.50132579459904e+09
  space :          3
  profit :  1.05039773837971e+10
  index to intpo, joinpo or selecpo :          5
  type of structure : int

  structure :          4
  density :  3.30213200840045e+09
  space :          4
  profit :  1.32085280336018e+10
  index to intpo, joinpo or selecpo :          9
  type of structure : int

  structure :          5
  density :  3.08224822441750e+09
  space :          8
  profit :  2.46579857953400e+10
  index to intpo, joinpo or selecpo :          1
  type of structure : int
```

```
structure :          6
density :  2.80442943898000e+09
 space :          4
profit :  1.12177177559200e+10
index to intpo, joinpo or selecpo :          6
type of structure : int

structure :          7
density :  2.42623937898000e+09
space :          5
profit :  1.21311968949000e+10
index to intpo, joinpo or selecpo :          7
type of structure : int

structure :          8
density :  1.80542615104235e+09
space :         26
profit :  4.69410799271011e+10
index to intpo, joinpo or selecpo :          3
type of structure : selecs
alternate no. :          7

structure :          9
density :  6.81088350705000e+08
space :          8
profit :  5.44870680564000e+09
index to intpo, joinpo or selecpo :          1
type of structure : selecs
alternate no. :          4

structure :         10
density :  5.89338260135509e+08
space :          6
profit :  3.53602956081305e+09
index to intpo, joinpo or selecpo :          2
type of structure : int

structure :         11
density :  4.80486898980000e+08
space :          2
profit :  9.60973797960000e+08
index to intpo, joinpo or selecpo :          4
type of structure : int
```

```
structure :          12
density :  4.51316901256487e+08
space :          51
profit :  2.30171619640808e+10
index to intpo, joinpo or selecpo :          2
type of structure : selecs
alternate no. :          7

structure :          13
density :  2.24602074630604e+07
space :        601
profit :  1.34985846852993e+10
index to intpo, joinpo or selecpo :          5
type of structure : joins
alternate no. :          2

structure :          14
density :  1.14105916421352e+07
space :        451
profit :  5.14617683060298e+09
index to intpo, joinpo or selecpo :          2
type of structure : joins
alternate no. :          2

structure :          15
density :  5.68535642949753e+06
space :        501
profit :  2.84836357117826e+09
index to intpo, joinpo or selecpo :          1
type of structure : joins
alternate no. :          2

structure :          16
density : -7.77511705623417e+07
space :        401
profit : -3.11782193954990e+10
index to intpo, joinpo or selecpo :          4
type of structure : joins
alternate no. :          2

structure :          17
density : -9.20304701230908e+07
space :        301
profit : -2.77011715070503e+10
index to intpo, joinpo or selecpo :          6
type of structure : joins
alternate no. :          2
```

```
structure :          18
density : -9.53140647096713e+07
space :          251
profit : -2.39238302421275e+10
index to intpo, joinpo or selecpo :          3
type of structure : joins
alternate no. :          2
```

12. THE RESULT OF THE POINTER ACCESS STRUCTURE OPTIMIZATION

```
space limit :          20

approximation tolerance :  5.00000000000000e-02

indexes of the chosen pointer access structures :

        5
        3
        2
        1

approximately optimal total profit :

   8.98888404135975e+10
```

13. COSTS OF PLACEMENT ACCESS STRUCTURES OBTAINED USING THE EXTRA VIEWS FROM STEP 12.

```
cluster :          1
  total cost :  9.09032718804036e+08
  number of views :          4

sorted :          1
  total cost :  1.83613021358218e+08
  number of views :          4
```

```
well-placed :          1
wlpl structure no:          1
  total cost :  1.41865965839206e+09
  number of views :          1
  associated relations :          1,          2

well-placed :          1
wlpl structure no:          2
  total cost :  1.89871926785093e+09
  number of views :          1
  associated relations :          1,          3

well-placed :          1
wlpl structure no:          3
  total cost :  7.57488670700074e+08
  number of views :          1
  associated relations :          1,          5

cluster :          2
  total cost :  3.00398973469964e+09
  number of views :          4

sorted :          2
  total cost :  6.61772032422955e+08
  number of views :          3

well-placed :          2
wlpl structure no:          1
  total cost :  2.21228725905313e+09
  number of views :          1
  associated relations :          2,          3

well-placed :          2
wlpl structure no:          2
  total cost :  7.32278058070598e+08
  number of views :          1
  associated relations :          2,          4

well-placed :          2
wlpl structure no:          3
  total cost :  1.39009466813195e+09
  number of views :          1
  associated relations :          2,          5

well-placed :          2
wlpl structure no:          4
  total cost :  1.41865965839206e+09
  number of views :          1
  associated relations :          1,          2
```

```
cluster :          3
  total cost :  1.13730584643508e+09
  number of views :          4

sorted :          3
  total cost :  2.88924643869908e+09
  number of views :          2

well-placed :          3
wlpl structure no:          1
  total cost :  1.24655183343679e+10
  number of views :          1
  associated relations :          3,          5

well-placed :          3
wlpl structure no:          2
  total cost :  1.89871926785093e+09
  number of views :          1
  associated relations :          1,          3

well-placed :          3
wlpl structure no:          3
  total cost :  2.21228725905313e+09
  number of views :          1
  associated relations :          2,          3

cluster :          4
  total cost :  9.80491611849854e+09
  number of views :          4

sorted :          4
  total cost :  2.57412228315277e+09
  number of views :          2

well-placed :          4
wlpl structure no:          1
  total cost :  1.82026470207064e+09
  number of views :          2
  associated relations :          4,          5

well-placed :          4
wlpl structure no:          2
  total cost :  7.32278058070598e+08
  number of views :          1
  associated relations :          2,          4

cluster :          5
  total cost :  2.05168755543343e+09
  number of views :          2
```

```
sorted :          5
  total cost :  1.32169747956371e+09
  number of views :          4

well-placed :          5
wlpl structure no:          1
  total cost :  7.57488670700074e+08
  number of views :          1
  associated relations :          1,          5

well-placed :          5
wlpl structure no:          2
  total cost :  1.39009466813195e+09
  number of views :          1
  associated relations :          2,          5

well-placed :          5
wlpl structure no:          3
  total cost :  1.24655183343679e+10
  number of views :          1
  associated relations :          3,          5

well-placed :          5
wlpl structure no:          4
  total cost :  1.82026470207064e+09
  number of views :          2
  associated relations :          4,          5
```

14. MINIMUM COST PLACEMENT ACCESS STRUCTURE CONFIGURATION

```
rel : 1,   type : well-placed with rel. 5

rel : 2,   type : well-placed with rel. 4

rel : 3,   type : cluster

rel : 4,   type : well-placed with rel. 2

rel : 5,   type : well-placed with rel. 1

minimum total cost : 2.626e+09
```

The optimal access structures are specified in steps II.12 and II.14. Specifically the access structures we obtain as a result of this design for the views given in step I.1 of the appendix are summarized in the following :

1. a well-placed access structure between relations 1 and 5,
2. a well-placed access structure between relations 2 and 4,
3. a cluster access structure on relation 3,
4. a pointer array access structure for processing intersection, difference and union views between relations 1 and 4,
5. a pointer array access structure for processing intersection, difference and union views between relations 3 and 4,
6. a pointer array access structure for processing intersection, difference and union views between relations 2 and 3,
7. a pointer array access structure for processing intersection, difference and union views between relations 1 and 2.

The total profits of the pointer access structures is 8.98e+10. The total cost of placement access struc-

tures is 2.98e+Ø9

The output of the metamorphosis is given at step II.2. Notice only a small portion of those access structures appear in the output of the optimization. This is mainly because of the small storage space limit. This restricts the number of pointer access structures that can be chosen. The storage space limit is kept low on purpose to reflect the various steps of the overall optimization algorithm well. Also the choice of access structures in the output of the optimization is mainly dependent on the cost functions used. In our example, this choice is arbitrary because of the arbitrary cost functions.