NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Previously copyrighted materials (journal articles, published tests, etc.) are not filmed.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

Les documents qui font déjà l'objet d'un droit d'auteur (articles de revue, tests publiés, etc.) ne sont pas microfilmés.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30.

Design of Distributed and Centralized Relational Databases

Hong Son Nguyen

A Major Technical Report

in

The Department

of

Computer Science

Presented in Partial Fulfillment of the Requirements
for the Degree of Master of Computer Science at
Concordia University
Montréal, Québec, Canada

December, 1987

ISBN 0-315-41604-1

ABSTRACT

Design of Distributed and Centralized Relational Databases

Hong Son Nguyen

This major report describes research techniques for designing distributed and centralized relational databases.

1. Various techniques for distribution design and integration of local relational databases are presented in detail. Some of these have been modified to adopt broader assumptions. Also, some have been implemented in C to automate their use (APPENDICES A and B). A brief look at the available commands of Distributed-INGRES and INGRES/STAR is also presented.

2. Logical design for relational databases consists of three phases : conceptual schema design, DBMS-independent logical design with the emphasis on integrity constraints, and DBMS-dependent logical design. Practical techniques for designing each of these phases are presented in detail.

3. DATAID technique for selecting secondary indices over the relations of a relational database is presented for physical design. The implementation of this technique in C has also been done to automate their use (APPENDIX C).

Practical guidelines for index selection from INFORMIX relational DBMS are also presented.

ACKNOWLEDGEMENTS

I would like to thank my supervisor, DR. P. Goyal, for his assistange and advice throughout this major technical report.

I also wish to thank DR. F. Sadri for reviewing this major report and giving me many valuable comments.

TABLE OF CONTENTS

chapter 1 DESIGN OF DISTRIBUTED DATABASES

ABSTRACT

A survey of the most recent research in distribution design and integration of local databases is conducted. Various design techniques, based on different design criteria, are presented in detail. Some of these techniques have been modified to be able to assume broader assumptions (section 1.3). Also, some have been implemented in C to automate their use. A brief look at the available commands of Distributed-INGRES (a pure distributed DBMS, which corresponds to distribution design) and INGRES/STAR (a multidatabase DBMS, which corresponds to integration of local databases) is also presented.

1.1. INTRODUCTION

A distributed database (DDB), as defined in [D19], is a set of data which is logically integrated, but physically distributed over the sites of a local or site-to-site computer network. Each site of the network can perform local transactions and must participate in at least one global transaction. A transaction is defined as local if it requires data stored at only one site, and global if it requires data stored at more than one site.

A distributed database management system (DDBMS) supports the creation and maintenance of DDBs. Its main objective is to provide the user with a uniform and integrated access to the data, while allowing the data to be physically distributed over the network sites; this is called distribution transparency. The degree of transparency supported can vary from system to system. The main types of transparency are [D10]:

- Fragmentation transparency, allowing the user to view a DDB as a set of global relations. This is the highest degree of transparency.

- Location transparency, requiring the user to work on fragments instead of global relations, but the location of fragments being envisible. This is a lower degree of transparency.

- Replication transparency, in which the user does not need to know the replication of fragments. Clearly, location transparency implies replication transparency; however, the reverse is not necessarily true,i.e. although the user does not need to know the replication of fragments, he must specify the location of a copy.

- Data model transparency, in which the user is not concerned with the data models of the underlying local

DBMSs.

DDBMSs can be classified into two types [D27]; **pure distributed database systems** (PDDBSs) and **multidatabase management systems** (MDBMSs). A PDDBS, such as Distributed-INGRES [D29], is suitable for DDBs which are to be developed from the scratch; such DDBs are called **pure distributed databases** (PDBs). A MDBMS, such as INGRES/STAR [D13], is suitable for DDBs which are developed as an integration of pre-existing local databases; such DDBs are called **multidatabases** (MDBs). In general, MDBs are probably more realistic than PDBs because an enterprise considering the development of a DDB may already have existing local databases, each of which is the result of an expensive investment, and furthermore, its local users are already familiar with it. Another distinguishing characteristic of a MDBMS is that each site has its own **local autonomy**, i.e. it decides what data to contribute to the MDB, what users can access that data, whether the contribution data can be updated, whether the logical and physical data structures of the contribution data can be changed, etc.

In this chapter, the design techniques for PDBs and MDBs will be presented in detail.

The design of a PDB, as indicated in [D10], consists of two phases :

1. Fragmentation design phase, which determines how global relations are split into horizontal, vertical, or mixed fragments. Horizontal fragmentation subdivides the tuples of a global relation into groups called horizontal fragments. Vertical fragmentation subdivides the attributes of a global relation into groups called vertical fragments. For both types of fragmentation, the following three rules must always be observed [D10]:

a. Completeness rule : all data in a global relation must be mapped into its fragments.

b. Reconstruction rule : a global relation can always be reconstructed losslessly from its fragments. In vertical fragmentation, the inclusion of the key of a given global relation into each of its fragments guarantees that this rule be observed.

c. Disjointness rule : the fragments of a global relation must be disjoint. This rule mainly applies to the horizontal fragmentation because the overlapping portion of two horizontal fragments cannot be known directly by the user, whereas data in the overlapping attributes of two vertical fragments can be exactly known.

2. Allocation design phase, which determines what network site(s) each fragment should be allocated. A fragment can be allocated nonreplicatedly, in which case it is allocated at only one site, or replicatedly, in which

case it is allocated at more than one site. Usually, replicated allocation is treated as a post-process of nonreplicated allocation.

The design of the global schema of a MDB consists of the following main steps [D27]:

1. Selection of a common semantic data model, such as the Entity-Category-Relationship data model (discussed later).

2. Translation of the local databases into their corresponding intermediate partial conceptual data models using the common data model selected in step 1.

3. Integration of the partial conceptual data models obtained in step 2 into an integrated one.

4. Finally, translation of the integrated conceptual data model obtained in step 3 into a set of global relation schemes.

The fragmentation design, allocation design, and integration of local databases have been investigated in various papers (from [D1] to [D30]). These research efforts have established both design criteria and mathematical foundations for future DDB design tools. They will be

presented in detail in the following sections.

1.2. DESIGN OF HORIZONTAL DISTRIBUTION

Horizontal distribution design consists of two phases : horizontal fragmentation design and horizontal allocation design.

1.2.1 Horizontal Fragmentation Design

1.2.1.1 HFD Method 1

In [D7,D10] a method which is based on the determination of a complete and minimal set of simple predicates has been proposed for horizontal fragmentation design. A predicate is called simple if it has the form :

attribute op value

where op can be =, !=, <, ≤, >, or ≥. A minterm predicate with respect to the set $P = \{p_i \mid 1 \le i \le n\}$ of simple predicates is the one which has the form : "$y = \bigwedge_{i=1}^{n} p_i$", where p_i is either p_i or $\neg p_i$. The set of all tuples for which y holds is called the minterm fragment corresponding to y. A simple predicate $p_i \in P$ is called relevant if there exist two minterm fragments whose corresponding minterm

predicates differ only in p_i (i.e. p_i in one of them and $\neg p_i$ in the other) such that they are accessed with different probabilities by at least one transaction.

P is complete if for each minterm fragment with respect to P all of its tuples are accessed with the same probability by any given transaction, and minimal if every simple predicate in P is relevant.

Now the procedure for determining a complete and minimal set P of simple predicates of a given relation R consists of the following steps :

1. Identify important transactions accessing R. A transaction is called important if it has a high execution frequency or requires a critical performance. This is due to 80/20 rule [D9] stating that the most active 20% of the transactions account for 80% of the total database usage. It is assumed that the same transaction which is issued at different sites is regarded as different.

2. Choose simple predicates on R, based on the processing locality of the transactions identified in step 1.

3. Apply the following algorithm :

P <- p_1, where p_1 is chosen such that it is relevant with respect to P (i.e. = {p_1})

while (P is not complete) {

P <- P U {p}, where p is chosen such that it is relevant with respect to new P (i.e. P with p just added)

P' <- set of nonrelevant predicates in P

/* the above step is necessary because a new predicate p can make other predicates in P to become nonrelevant */

P <- P-P'

}

The result is a complete and minimal set P of simple predicates. The minterm fragments produced from this set P can be regarded as "appropriate" units for data allocation because of the following reasons :

a. Each minterm fragment is accessed homogeneously by any transaction (due to completeness property of P).

b. Any two minterm fragments differing in only one simple predicate are accessed dishomogeneously by at least one transaction (due to relevance property of P).

c. P takes into account the processing locality of the important transactions accessing R (due to steps 1 and 2).

1.2.1.2 HFD Method 2

Another method [D9], which is based on the minimization of both access costs at local sites and data transmission costs between sites, has also been proposed for horizontal fragmentation design. First, it is necessary to distinguish between primary and derived horizontal fragmentation. A horizontal fragmentation $(q_1,\ldots,q_n)$ of a global relation R is called primary if it is based on some property of its own attributes; it is called derived if there exists a horizontal fragmentation $(q'_1,\ldots,q'_n)$ (primary or derived) of another relation S and a join predicate jp such that $q_i = R\ SJ_{jp}\ q'_i$, where SJ is the semi-join and $1 \leq i \leq n$. Second, the method makes the following assumptions :

1. Each global relation may have zero or more candidate fragmentations, each of which is primary on that relation and derived on zero or more other relations connected to it via joins. These candidate fragmentations are specified in advance by the user.

2. For each horizontal fragment q of a primary frgmentation p on a relation i, the allocation site of q is specified in advance by the user.

3. Each horizontal fragment q of a derived

fragmentation p on a relation i must have the same allocation site as the corresponding fragment q' of a relation i' fragmented by p (primary or derived) and connected to i via a join.

4. For each site j, CLR_j, CLU_j, CRR_j, and CRU_j denote the costs of a local retrieval access, local update access, remote retrieval access, and remote update access, respectively; these access costs may have different values. Each access is assumed to retrieve or update one tuple.

5. The transmission cost TC of a byte between any two different sites is assumed to be a constant.

In the solution, each relation is either (1) fragmented by the best candidate fragmentation or (2) not fragmented at all and allocated as a whole at the best site. The objective function to determine the best candidate fragmentations and the best sites is as follows :

$$\min z = \sum_{i,p} C_{ip} * X_{ip} + \sum_{i,j} D_{ij} * Y_{ij} - \sum_{h,p} A_{hp} * W_{hp} - \sum_{h,j} B_h * V_{hj}$$

where :

1. X_{ip} = 1 if p is selected for i
   0 otherwise

C_{ip} is the corresponding cost due to the fragmentation of i by p. C_{ip} consists of an access component CA_{ip} and a transmission component CT_{ip}, i.e. :

$$C_{ip} = CA_{ip} + CT_{ip}$$

a. CA_{ip} is computed as follows :

$$CA_{ip} = \sum_{k} SCA_{ipk}$$

where SCA_{ipk} is the access cost for a single transaction k per unit time, and is computed as follows :

$$SCA_{ipk} = [\sum_{j \neq o(k)} (NR_{ipj} * CRR_j + NU_{ipj} * CRU_j)] + (NR_{ip,o(k)} * CLR_{o(k)} + NU_{ip,o(k)} * CLU_{o(k)})$$

where NR_{ipj} and NU_{ipj} are the number of retrieval accesses and update accesses per unit time, respectively, made by transaction k to the fragment of relation i fragmented by p and allocated at site j ; o(k) is the site of origin of k.

b. CT_{ip} is computed as follows :

$$CT_{ip} = CT'_{ip} + CT''_{ip}$$

- CT'_{ip} is computed as :

$$CT'_{ip} = \sum_{h} TR_{hi} * N_p * TC$$

where TR_{hi} is the number of bytes of the join columns and the result columns which must be transmitted per unit time to each of N_p fragments of relation i (fragmented by p) by all transactions which perform join h from some other relation i' to i.

- CT''_{ip} is computed as :

$$CT''_{ip} = \sum_{k} SCT_{ipk}$$

where SCT_{ipk} is the cost of transmitting the result of transaction k from allocation sites of fragments of i to the site of origin of k per unit time; k belongs to the set of transactions accessing i.

2. Y_{ij} = 1 if i is allocated at j as a whole
   0 otherwise

D_{ij} is the corresponding cost due to the allocation of i at j as a whole. D_{ij} is computed in the same way as C_{ip} in which i is considered as the only fragment of i allocated at j.

3. W_{hp} = 1 if p is used for deriving a fragmentation of some relation i from

another relation i' via h.

0 otherwise

A_{hp} is the corresponding savings due to the fact that the join h between two relations i and i' can now be performed locally at the corresponding pairs of fragment sites. A_{hp} is computed as follows :

$$A_{hp} = (TR_{hi} + TR_{hi'})*N_p*TC$$

where TR_{hi} and $TR_{hi'}$ are defined as above, in which TR_{hi} is applied to all transactions that perform join h from i' to i, whereas $TR_{hi'}$ is applied to all transactions that perform join h from i to i'.

4. V_{hj} = 1 if h is used to join two relations, say i and i', allocated at j as a whole

0 otherwise

B_h is the corresponding savings due to the fact that the join h of i and i' can now be performed locally at one site. B_h is assumed not to depend on the choice of site j. B_h is computed in the same way as A_{hp} (just as D_{ij} is computed in the same way as C_{ip}), i.e. :

$$B_h = (TR_{hi} + TR_{hi'})*TC$$

Note that in the solution, each relation i has been either (1) fragmented by the best fragmentation p selected by the method, in which case the allocation sites of the corresponding fragments of i are those specified in advance by the designer (see assumption 2 above), or (2) allocated as a whole at some best site j selected by the method. Thus the method not only solves the horizontal fragmentation problem but also solves the horizontal allocation problem.

1.2.2 Horizontal Allocation Design

1.2.2.1 HAD Method 1

In [D10] a method which is based on maximum processing locality of transactions has been proposed for the horizontal allocation design. Let r_{ij} and u_{ij} denote the number of retrieval accesses and update accesses, respectively, to fragment i by all transactions originated at site j per unit time.

1. In the non-replicated allocation design, fragment i is allocated at site j^* where it is accessed most, i.e.:

$$n_{ij^*} = r_{ij^*} + c \cdot u_{ij^*} \text{ is maximum}$$

where c is a constant which is the ratio between update and retrieval access costs. Constant c is assumed to be

independent of sites, remoteness, and locality of accesses.

2. In the replicated allocation design, fragment i is allocated at site j if the benefits for retrieval transactions originated at j which finds i as a local copy is greater than the costs of updating this copy from all other sites to preserve data consistency, i.e.:

$$d_{ij} = r_{ij} - \sum_{j' \neq j} c^* u_{ij'} > 0$$

If for every site j, $d_{ij} \leq 0$, then i will be allocated at j* such that d_{ij^*} is maximum.

See Appendix A for the implementation in C of this method.

1.2.2.2 HAD Method 2

Another method [D9], based on the non-replicated allocation solution from the HFD Method 2 (see the note at the end of section 1.2.1.2) and greedy heuristics has also been proposed for the replicated allocation design. In the solution, a global relation can be allocated according to multiple candidate horizontal fragmentations or be stored as a whole at multiple sites , or both. The method works as follows :

Let S and C denote the current replicated allocation

solution and its corresponding total cost, respectively; X denote either X_{ij} or Y_{ij}. Now S and C start with S* and C*, respectively, where S* and C* are the nonreplicated allocation solution and its corresponding total cost. For each X which has the value 0 in S, build S' = S ∪ { X =1 } and compute its corresponding total costs as follows :

$$C' = \sum_{k \notin T(i)} TEC(S,k) + \sum_{k \in T(i)} TEC(S',k)$$

where T(i) is the set of transactions which access relation i, and TEC(S,k) is the transaction execution cost for transaction k in the solution S; TEC(S,k) is computed by the optimizer of a particular relational DBMS, and thus is DBMS-dependent. Note that in the formula the execution costs of those transactions which access relation i must be recomputed in the new solution S', whereas those transactions which do not access i can reuse their old execution costs in the old solution S. Let X* denote the X which corresponds to minimum C'. If that minimum C' is less than the current C, then X* is added to S and C is set to that minimum C'. Repeat the algorithm with new S and C. The algorithm stops when a minimum C' is found which is greater than or equal to the current C. The method is classified as heuristic because at each iteration a variable X is selected which results in minimum total cost.

1.3. DESIGN OF VERTICAL DISTRIBUTION

As in horizontal distribution design, vertical distribution design also consists of two phases : vertical fragmentation design and vertical allocation design.

1.3.1 Vertical Frgmentation Design

In [D24] a method which is based on the concept of attribute affinity has been proposed for vertical fragmentation design. First, two types of vertical fragmentation are distinguished: vertical partitioning in which each nonkey attribute belongs to only one vertical fragment, and vertical clustering in which a nonkey attribute may belong to more than one vertical fragment. The key attribute of the relation must be present in each vertical fragment in both cases to guarantee the lossless decomposition property. Now, let R be the global relation to be vertically fragmented, $A=\{a_i \mid i = 1,\ldots,n\}$ be the set of all attributes of R, T be the set of all transactions accessing R, each of which may be originated at multiple sites. Unlike in [D24] where all attributes accessed by each transaction are assumed to be either retrieved or updated by that transaction, it is assumed in the following that a transaction may both retrieve some and update the others of its accessed attributes; as a result, some of the formulations in [D24] will be modified in accordance with

this broader assumption. For each transaction k in T, the following variables are defined :

use_{ki} = 1 if k uses attribute a_i
0 otherwise

$type_k$ = 'r' if k is a retrieval transaction
'u' if k is an update transaction

acc_k = total number of accesses to R made by k per unit time (each access retrieves or update one tuple; k may be originated from multiple sites)

$A(k) = \{a_i \mid a_i \in A \wedge use_{ki}=1\}$

(i.e. A(k) is the set of attributes of relation R accessed by k).

The affinity of a_i and a_j, written aff_{ij}, is defined as the number of accesses to R made by all transactions which use a_i and a_j together, i.e. :

$$aff_{ij} = \sum_{(use_{ki}=use_{kj}=1)} acc_k$$

The affinities are recorded in a matrix, called attribute affinity (AA) matrix. The AA matrix is symmetric and semiblock (i.e. each diagonal element is greater than any other element on the same row or column). Now the rows and

columns of AA are permuted such that the following expression is maximized :

$$\sum_{i,j} aff_{i,j} * (aff_{i,j-1} + aff_{i,j+1} + aff_{i-1,j} + aff_{i-1,j})$$

where $aff_{i,0} = aff_{0,j} = aff_{i,n+1} = aff_{n+1,j} = 0$.

The result matrix is called the clustered AA matrix. It is also symmetric and semiblock [D24].

1.3.1.1 Vertical Partitioning Design

Let x be a point along the main diagonal of the clustered AA matrix. x divides R into two vertical fragments : U consisting of the attributes from the first attribute up to x, and L from x to the last attribute. LT, UT, and IT denote sets of transactions which access only L, only U, and both L and U, respectively, i.e. :

$$LT = \{k \mid k \in T \wedge A(k) \subseteq L\}$$

$$UT = \{k \mid k \in T \wedge A(k) \subseteq U\}$$

$$IT = T - (LT \cup UT)$$

Now compute :

$$CL = \sum_{k \in LT} acc_k; \qquad CU = \sum_{k \in UT} acc_k; \qquad CI = \sum_{k \in IT} acc_k$$

Choose x_{max} such that the objective function

$z = CL*CU-CI^2$ is maximized.

Then the vertical partitioning which corresponds to x_{max} is accepted if $z > 0$, and rejected otherwise.

1.3.1.2 Vertical Clustering Design

Let x_1 and x_2 be two points along the main diagonal of the clustered AA matrix. x_1 and x_2 divide R into two vertical fragments : U consisting of the attributes from the first one up to x_2, and L from x_1 to the last one. Let I be the set of attributes between x_1 and x_2, i.e. $I = L \cap U$.

The same objective function z as in section 1.3.1.1 is used for choosing x_1 and x_2, i.e.:

$z = CL*CU-CI^2$ is maximized,

but LT and UT are now defined as follows :

$$LT = \{k \in T \mid (type_k = 'r' \wedge A(k) \subseteq L) \vee (type_k = 'u' \wedge$$

$$A(k) \subseteq L \wedge A_u(k) \cap I = \emptyset)]$$

$$UT = \{k \in T \mid (type_k = 'r' \wedge A(k) \subseteq U) \vee (type_k = 'u' \wedge A(k) \subseteq U \wedge A_u(k) \cap I = \emptyset)\}$$

where $A_u(k)$ denotes the set of attributes updated by k.

IT is defined as in section 1.3.1.1, i.e. :

$$IT = T - (LT \cup UT)$$

1.3.2 Vertical Allocation Design

The following methods are used for non-replicated allocation of vertical fragments.

1.3.2.1 NAVF Method 1

Let c_1, c_2, and c_3 be the access cost (retrieval and update access costs are assumed to be the same for simplicity), storage cost per byte, and transmission cost per byte, respectively. Let $alloc_F$ denote the set of allocation sites for fragment F, which may consist of either a single site in the nonreplicated allocation design or multiple sites in the replicated one. In [D24] a method, which is based on the minimization of the total access costs, storage costs, and transmission costs, has been proposed for the nonreplicated vertical allocation design. Suppose that a

global relation R is vertically partitioned into L and U, then :

1. The number of accesses made by all transactions to only L, only U, or both L and U is as follows :

$$W_1 = CL + CU + 2*CI$$

where CL, CU, and CI are defined as in section 1.3.1.1.

2. The number of memory bytes which are replicated in both L and U, except the required common key attribute, is:

$W_2 = 0$ (because R is vertically partitioned)

3. The total number of bytes which are transmitted from remote sites to the sites of origin of all transactions is:

$$W_3 = \sum_k \sum_{k \in S_k} acc_k^{\overline{alloc}_F} \sum_{a_i \in F \cap A(k)} l_i$$

where S_k is {L}, {U}, or {L,U}, depending on whether $k \in LT$, $k \in UT$, or $k \in IT$, respectively; l_i is the length of attribute a_i; $acc_k^{\overline{alloc}_F}$ is the total number of accesses to F made by transaction k originated from all sites other than the allocation site $alloc_F$ (which is a singleton in this non-replicated case) per unit time. Choose $alloc_F$ and

$alloc_u$ such that the objective function :

$$z = \sum_{i=1}^{3} c_i * W_i \text{ is minimum}$$

where c_1, c_2, c_3 are defined as above.

See Appendix B for the implementation of this method.

1.3.2.2 NAVF Method 2

Another approach [D10] which is based on the maximum processing locality of transactions has also been proposed. When a fragment F allocated at site r is vertically partitioned into two fragments F_1 and F_2 allocated at sites s and t, respectively, the benefit $B_{r,s,t}$ and the cost $C_{r,s,t}$ of this allocation is computed as follows :

$$1.\ B_{r,s,t} = \sum_{k \in T_1} acc_k(s) + \sum_{k \in T_2} acc_k(t)$$

where $acc_k(s)$ is the number of accesses to F made by transaction k originating from site s; similarly for $acc_k(t)$; T_1 and T_2 are defined as follows :

$$T_1 = \{k \in T \mid A(k) \subseteq F_1\}$$

$$T_2 = \{k \in T \mid A(k) \subseteq F_2\}$$

(i.e. T_1 is the set of transactions accessing only F_1, and T_2 is the set of transactions accessing only F_2).

$$2.\ C_{rst} = \sum_{k \in T_3} acc_k(r) + 2 * \sum_{k \in T_4} acc_k(r) + \sum_{j \neq r,s,t} \sum_{k \in T_4} acc_k(j)$$

where

$$T_3 = \{k \in T \mid A(k) \subseteq F_1 \vee A(k) \subseteq F_2\}$$

$$T_4 = T - T_3$$

(i.e. T_3 is the set of transactions which access either F_1 or F_2, while T_4 is the set of transactions which access both F_1 and F_2).

The savings S_{rst} of this allocation is :

$$S_{rst} = B_{rst} - C_{rst}$$

Choose s and t such that S_{rst} is maximized.

See Appendix A for the implementation in C of this method.

1.3.2.3 NAVF Method 3

In [D10] another method has also been proposed for the nonreplicated allocation of vertically clustered fragments. B_{rst}, C_{rst}, and S_{rst} are defined as above, but T_1, T_2, T_3, and T_4 are defined differently as follows :

$$T_1 = \{k \in T \mid (type_k = 'r' \wedge A(k) \subseteq F_1) \vee (type_k = 'u' \wedge A(k) \subseteq F_1 \wedge A_u(k) \cap I = \emptyset)\}$$

(i.e. T_1 is the set of transactions accessing only F_1).

$$T_2 = \{k \in T \mid (type_k = 'r' \wedge A(k) \subseteq F_2) \vee (type_k = 'u' \wedge A(k) \subseteq F_2 \wedge A_u(k) \cap I = \emptyset)\}$$

(i.e. T_2 is the set of transactions accessing only F_2).

$$T_3 = \{k \in T \mid type_k = 'r' \wedge (A(k) \subseteq F_1 \vee A(k) \subseteq F_2)\} \cup \{k \in T \mid type_k = 'u' \wedge [(A(k) \subseteq F_1 \vee A(k) \subseteq F_2) \wedge A_u(k) \cap I = \emptyset]\}$$

where $A_u(k)$, again, denotes the set of attributes in F updated by transaction k.

(i.e. T_3 is the set of transactions accessing either F_1 or F_2).

$$T_4 = T - T_3$$

(i.e. T_4 is the set of transactions accessing both F_1 and F_2).

Now let's turn to the replicated allocation of vertical fragments. Let $alloc_F$ be the nonreplicated allocation site of vertical fragment F obtained from the NAVF method 1 in section 1.3.2.1. The set of additional sites at which F can be allocated is determined by the following algorithm [D24]:

repeat

try all sites not in $alloc_F$

let j be the site which results in the maximum benefit

$alloc_F \leftarrow alloc_F \cup \{j\}$

until no further benefit

Site j is the most beneficial if the following objective function

$$z = \sum_{i=1}^{3} c_i * W_i \text{ is minimum}$$

where :

1. $$W_1 = \sum_{k \mid type_k = 'u' \wedge A_u(k) \cap F \neq \phi} (|alloc_F|) * acc_k$$

where $|alloc_F|$ is the number of copies of F. The meaning of W_1 is that it is the number of update accesses to be applied to all the copies of F to make them consistent.

2. $$W_2 = |alloc_F| * |F| * \left(\sum_{a_i \in F} l_i \right)$$

where |F| is the number of tuples in F. The meaning of W_2 is that it is the number of memory bytes for all the copies of F.

$$3.\ W_3 = \sum_k acc_k^{\overline{alloc}_F} \sum_{a_i \in F \cap A(k)} l_i$$

where $acc_k^{\overline{alloc}_F}$ is, as before, is the total number of accesses to F made by transaction k originated from all sites other than those in $alloc_F$. The meaning of W_3 is that it is the total number of bytes which must be transmitted to the sites of origin of all transactions.

c_1, c_2, c_3 are defined as in section 1.3.2.1.

Note that when the number of copies of F increases, W_1 and W_2 also tend to increase, whereas W_3 tends to decrease. The algorithm can be classified as heuristic because at each iteration the most beneficial site is selected for replication.

1.4. ALLOCATION OF FRAGMENTS IN A LOCAL AREA NETWORK

A different approach to data allocation which takes into account the characteristics of a local area network has been proposed in [D25]. The method works as follows :

Let $P = \{P_1,\ldots,P_p\}$ be the set of processor nodes, CC_i and CI_i be the processing and I/O capacities of P_i, respectively, and CM_{ij} be the communication capacity between P_i and P_j; $F = \{F_1,\ldots,F_z\}$ the set of fragments to

be allocated over P; $T = \{T_1,\ldots,T_t\}$ the set of transactions operating on F. The method is different from those in site-to-site networks in that the set of objects to be allocated to P includes both fragments and transactions, i.e. :

$$O = F \cup T = \{O_1,\ldots,O_n\} \text{ (say)}$$

The cost of allocating O to P is computed as :

$$tc = \sum_{i=1}^{p} (vi_i + vc_i) + \sum_{i=1}^{p-1} \sum_{j=i+1}^{p} vm_{ij}$$

where vi_i, vc_i are the I/O and processing load at P_i, respectively; vm_{ij} is the comunication load between P_i and P_j, $i \neq j$. The computation of vi_i, vc_i, vm_{ij} is DBMS-dependent.

The objective is to allocate O to P such that tc is minimized and for every P_i, P_j in P, $vi_i \leq CI_i$, $vc_i \leq CC_i$ and $vm_{ij} \leq CM_{ij}$. A greedy first-fit algorithm is presented in [D25] to solve this proplem. It works as follows :

The algorithm starts with O and D = ∅ (D is a set in which each of its elements is a collection of objects in O; the meaning of D will become clear in the algorithm), and a set $N = \{N_1,\ldots,N_{n-1}\}$ of n-1 fictitious nodes where n is the number of objects in O. For every $(O_i,O_j) \in O^2$, $O_i \neq O_j$,

allocate $\{O_i, O_j\}$ to one node and each object in $O-\{O_i, O_j\}$ to each of the remaining fictitious nodes. Compute tc for this allocation. Let (Ob_1, Ob_2) be the pair of objects such that tc is minimum and the constraints on processor nodes are satisfied (in the constraints it is assumed that $N_i = P_i$, $i = 1,\ldots,p$; the constraints are specified by the designer). Set $D = D \cup \{\{Ob_1, Ob_2\}\}$, $O = O - \{Ob_1, Ob_2\}$, and reduce the number of fictitious nodes by 1. Repeat the above steps until the constraints in the first computation of tc in an iteration for determining (Ob_1, Ob_2) are not satisfied. If O becomes $\emptyset$, then D is the allocation solution of the initial O to P. In any iteration after the first one, the following rules must be followed :

1. If the number of elements in D (called compound object) reaches p, then at least one of O_i and O_j must be taken from D.

2. If exactly one of Ob_1 and Ob_2 is a compound object, say Ob_1, then O and D are modified at the end of the corresponding iteration as follows :

$$O = O - \{Ob_2\}$$

$$D = (D - \{Ob_1\}) \cup \{Ob_1 \cup \{Ob_2\}\}$$

3. If both Ob_1 and Ob_2 are compound objects, then O

and D are modified at the end of the corresponding iteration as follows :

$$O = O$$

$$D = (D - \{Ob_1, Ob_2\}) \cup \{Ob_1 \cup Ob_2\}$$

The result of applying the above rules is that the number of compound objects (i.e. the cardinality of D) is never greater than p.

1.5. GLOBAL SCHEMA DESIGN FOR MULTIDATABASES

The main problems associated with the global schema design for a multidatabase (MDB) will be discussed in this section.

1.5.1 Selection of a Common Data Model

A suitable common data model must provide for the concept of generalization because this concept is essential when discussing integration of multiple databases. For example, the Entity-Category-Relationship (E-C-R) data model [D16,D17,D18,D23] is one of such models. This model provides two additional concepts for the Entity-Relationship (E-R) model. First is the concept of category. There are two types of categories : the

generalization category groups entities from several entity sets or categories when these entities participate in the same relationship, and the subset category groups entites from one entity set or category when these entities have specific attributes which are not shared by the rest. Second, for each entity set or category participating in a relationship set, two cardinality constraints, called mincard and maxcard, are used to specify the minimum and maximum number of relationship instances per instance of that entity set or category; a mincard 0 implies a partial participation, and 1 a total participation. Also, a role name can be specified for every entity set or category participating in a relationship set. Entity sets, relationship sets, and attributes are defined as in the E-R model. The term object set will be used in the following to refer to an entity set or a category.

In the E-C-R diagram, rectangular boxes represent entity sets, diamond-shaped boxes relationship sets, and hexagonal boxes categories. See [D17] for an example.

In the following, it is assumed that the conversion of local databases to the common E-C-R data model has been done. After this, integration of the partial conceptual E-C-R models in to an integrated one consists of two phases:

1. Pre-integration phase, in which the semantics of the partial E-C-R models are analysed to identify similarities and conflicts; and in the case of conflicts, how to resolve them.

2. Integration phase, which integrates similar object sets and relationship sets in the partial conceptual E-C-R models, based on the results obtained from the pre-integration phase.

1.5.2 Pre-integration

During the semantic analysis of the partial E-C-R models, the following types of conflict are identified [D14] : naming conflicts (i.e. synonyms and homonyms), scaling conflicts (i.e. using different scales to measure the same data item), representation conflicts (i.e. using differing data types to represent the same data item), and structural conflicts (i.e. to represent a real-world object by different data structures, e.g. as an entity set in one database, as a relationship set in another, or as an attribute in another, etc.).

Naming conflicts are resolved by renaming. Scaling conflicts or representation conflicts are resolved by using a common scale or data type and converting the other scales or data types into this common scale or data type.

Resolution of structural conflicts will be discussed in section 1.5.3.

In integrating attributes, the concepts of local equivalence and global equivalence must be understood first [D15,D21]. Two semantically equivalent key attributes are globally equivalent if the uniqueness property still holds in the union of entities; otherwise, they are locally equivalent. For example, Social Insurance Number (SIN) is a globally equivalent key because each employee has a unique SIN regardless of which company he works, whereas Employee Number is a locally equivalent key because two different employees working in two different companies may have the same employee number. Two semantically equivalent nonkey attributes are globally equivalent if the same entity must have the same values for the two attributes; otherwise, they are locally equivalent. For example, Age is a globally equivalent nonkey attribute because the same employee must have the same age, whereas Salary is a locally equivalent nonkey attribute because the same employee may work in two different companies at the same time and thus may have two different salaries.

For locally equivalent key attributes, three solutions are possible :

1. Add attributes to one or more local databases such

that there will be a globally equivalent key among them.

2. Add a new attribute in the global database and convert existing local key values into global key values.

3. Prefix the locally equivalent keys with a local database identifier to make them unique.

Solutions (1) and (2) are suitable when the intersection of the object sets involved is non-empty; otherwise, solution (3) is suitable.

For locally equivalent nonkey attributes, two solutions are possible :

1. If the intersection of the object sets involved is empty, then create a global attribute whose values are taken from the local attributes; otherwise,

2. Create a new global attribute whose values are computed from the local attributes in some way (i.e. average, sum, etc.).

1.5.3 Integration of Local Databases

The integration of attributes of similar object sets or relationship sets has been discussed in section 1.5.2 above

(using the concept of attribute equivalence). The integration of similar object sets or relationship sets will be presented in this section.

For integration of similar object sets, the union of all object sets in all local E-C-R models are divided into disjoint subsets, each of which represents a similar type of information.

Let $A_1,\ldots,A_n$ be similar object sets to be integrated. Then the integration of $A_1,\ldots,A_n$ is governed by the following rules :

1. If $A_1=\ldots=A_n$, then a single object set A similar to A_i $(i = 1, n)$ is created in the integrated E-C-R schema.

2. If $A_1 \subseteq A_2 \subseteq \ldots \subseteq A_n$, then the integrated E-C-R schema consists of the same object sets in which A_{n-1} is a subset category of A_n, A_{n-2} is a subset category of A_{n-1}, and so on (Fig.1, [D18]).

3. If $A_i \cap A_j \neq \emptyset$ and $A_i \not\subseteq A_j$ and $A_j \not\subseteq A_i$ for every $i \neq j$, then the integrated E-C-R schema consists of 2^n-1 object sets in the most general case. If for some i^*, j^*, $A_{i^*} \cap A_{j^*} = \emptyset$, then A_{i^*} and A_{j^*} will not be integrated (Fig.2 for the case n=3, [D18]).

4. If $A_i \cap A_j = \emptyset$ for every i != j, then the decision of whether to integrate all of the A_is, some of the A_is, or none of them is left to the designer. The technique used to integrate all of them is the same as in the case 3 above but with $A_i \cap A_j = \emptyset$ for every i != j (Fig.3 is for this case, [D18]).

The integration of similar relationship sets depends on a number of factors : (i) degree, which is defined as the number of object sets participating in the relationship set; (ii) role of each object set participating in the relationship set; (iii) cardinality constraints (defined in 1.5.1), and (iv) semantics of data. It is governed by the following rules :

1. The relationship set with a higher degree should always be retained in the integrated schema.

Example 1 [D23] : Let R be a relationship set involving two object sets A and B with primary keys a# and b#, respectively. Let C be an object set with a# and b# being two of its attributes (C can be considered as a zero-degree relashionship set). Now the integration of R and C is possible with relationship R being retained if :

- R is m-to-n and (a#,b#) is the key of C.

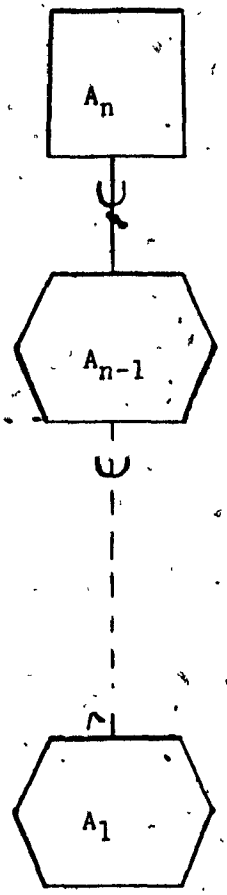


Fig. 1

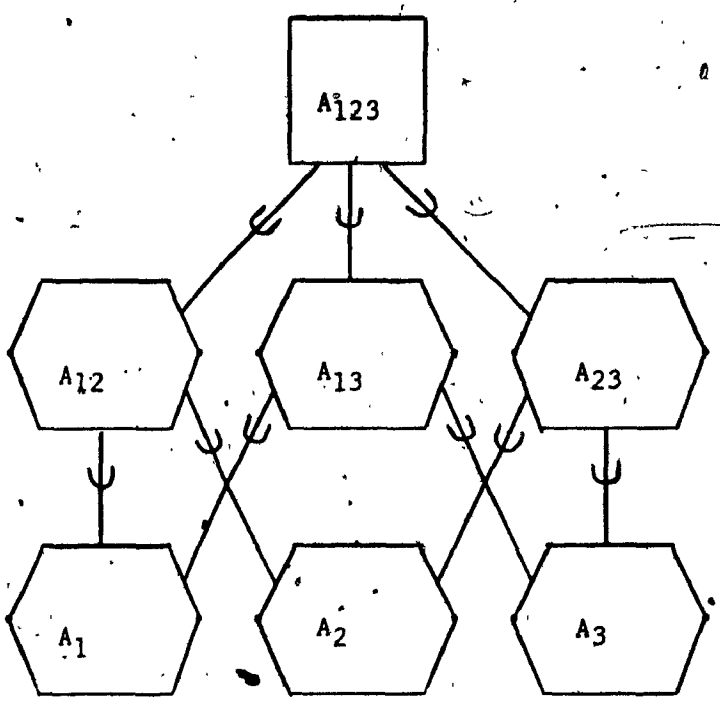


Fig. 2

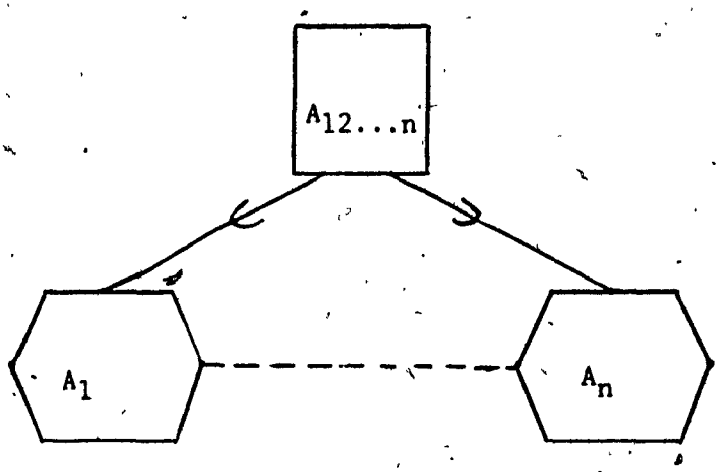


Fig. 3

- R is m-to-1 between A and B, and the key of C is a#.

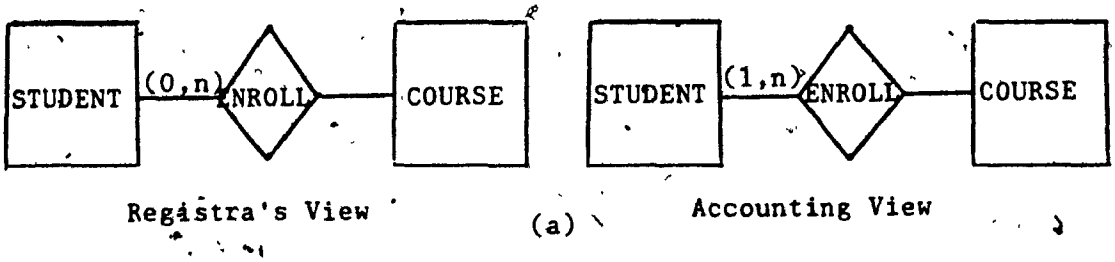
- R is 1-to-n between A and B, and the key of C is b#.

The integration is possible because the above conditions allows to derive C from R (i.e. R and C have the same meaning).

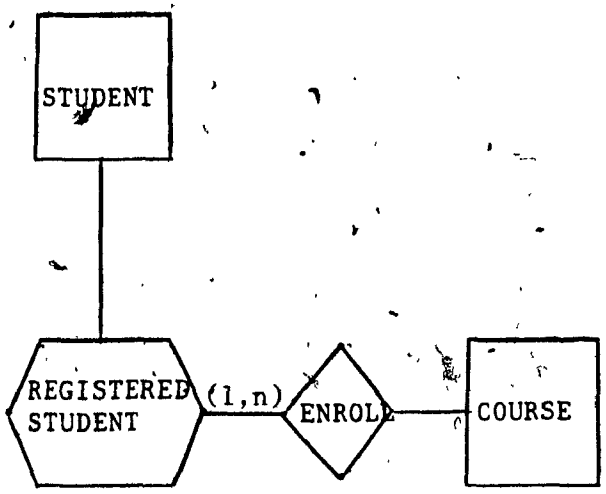
Example 2 [D23] : Let R_1 be a relationship set between A, B, and C; c# be the key of C; R_2 be a relationship set between A and B with c# being one of its attributes. Then R_2 can be derived from R_1, and R_1 is retained in the integrated schema if the relationship between A anb B is m-to-n in R_2 and the relationship between (A,B) and C is m-to-1 in R_1. An example can be found in [D23].

2. The relationship set with a total participation should always be retained in the integrated schema.

Example 3 [D23] : Consider the two relationships in Fig.4(a). The first relationship is from the Registra's Office where some students may not enroll in any courses, whereas the second one is from the Accounting Office which is concerned with only enrolled students. The relationship integration will retain the second one and the result is shown in Fig.4(b).



Registra's View (a) Accounting View



(b)

Fig. 4

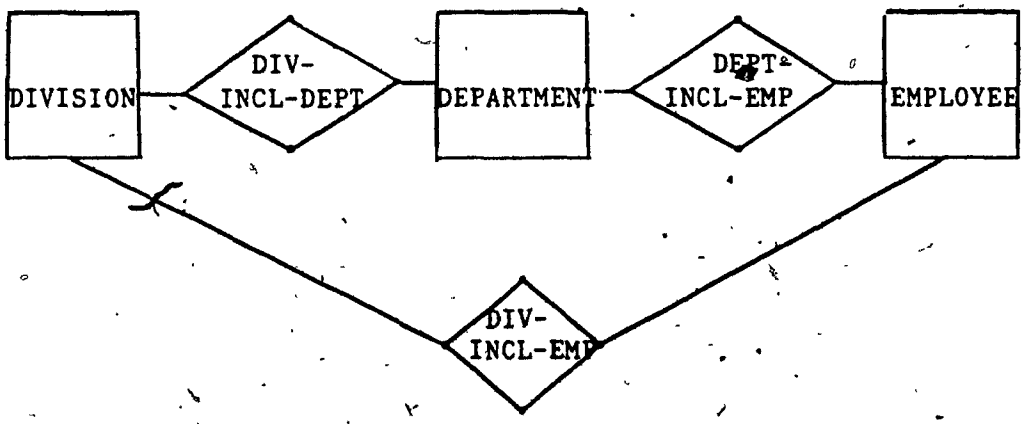


Fig. 5

3. When deriving a relationship set from several other relationship sets, always consider the semantics of data; otherwise, a connection trap may occur.

Example 4 [D30] : Consider the E-C-R diagram in Fig.5. Relationship Div-Includes-Emp cannot be derived from Div-Includes-Dept and Dept-Includes-Emp if some emplyees such as technical advisors are considered to belong directly to divisions, but not to departments

4. The integration of similar object sets participating in the relationship set is governed by the same object integration rules as above.

1.6. DISTRIBUTED-INGRES AND INGRES/STAR

Distributed-INGRES [D29] is a PDDBS developed at University of California at Berkeley. The system allows the user to view a DDB as a collection of global relations, thus fragmentation and location transparency are provided. Horizontal fragmentation is supported, but vertical one is not. Replication of fragments is allowed. The following Distributed-INGRES commands are provided :

1. dcreatedb ddb-name at site-list

2. expand ddb-name at site-list

3. drop ddb-name at site-list

4. dingres ddb-name

5. dcreate rel-name (col-name = data-type [, col-name
   data-type] ...) at site-list

6. distribute rel-name
   at site-list where p
   [, at site-list where p]

7. distribute rel-name at site-name

8. extend rel-name to site-list

9. destroy rel-name at site-list

A brief explanation of these commands is as follows : a DDB can be created at a set of sites by a dcreatedb command. The set of sites where a DDB exists can be expanded or shrunk by an expand or drop command, respectively. A dingres command invokes Distributed-INGRES for an existing DDB; that DDB now becomes current, and any normal QUEL command is executed on that DDB. A new relation can be created at a set of sites, which must be a subset of the sites where the current DDB exists, by a dcreate command.

The set of sites where a relation exists can be expanded or shrunk by an **extend** or **destroy** command, respectively. Finally, a relation can be fragmented and distributed over the sites where it exists, or distributed as a whole at one site by a **distribute** command.

INGRES/STAR [D13] is a MDBMS developed at Relational Technology Inc.. The system supports transparency over a variety of different CPUs and operating systems connected via a variety of different communication networks. At present, each local relational DBMS must be a centralized INGRES, but Relational Technology Inc. plans to support non-INGRES sites in the near future. The following INGRES/STAR commands are provided :

1. **createdb** ddb-name

2. **define link** drel-name
   with node = site-name
   database = ldb-name
   table = lrel-name

3. **destroy link** drel-name

4. **destroydb** ddb-name

5. **add node** site-name

6. **remove node** site-name

A brief explanation of these commands is as follows : A distributed database can be created at a site, say X, by a **createdb** command (X is the site where the **createdb** command is issued); now the database can be referenced at that site. However, it can be made directly accessible from additional sites Y, Z, etc. by a number of **add node** commands. Added nodes can later be removed by a **remove node** command. A MDB can be destroyed from any of its directly accessible sites by a **destroydb** command. A global relation can be created at one of the accessible sites by a **define link** command, and be destroyed by a **destroy link** command.

After the MDB has been created, it can be accessed by any normal DML command, as if it were a local database.

1.7. CONCLUSION

There are two approaches to distributed database design, namely, distribution design and integration of local databases. The first approach is suitable for distributed databases which are designed from scratch, whereas the second approach is suitable for designing a logically integrated database from several local databases. Various techniques for both approaches have been described; some of

them have been implemented in C to automate their use. These techniques have helped us understand both mathematical aspects and design criteria for designing distributed databases. These can also help automate the design process in the future.

Chapter 2 LOGICAL DESIGN FOR RELATIONAL DATABASES

ABSTRACT

Logical design for relational databases consists of three phases : conceptual schema design, DBMS-independent logical design with the emphasis on integrity constraints, and DBMS-dependent logical design. A survey of research techniques for designing each of these phases is conducted. Some of these techniques have been modified to be able to apply them to the selected environment.

2.1. INTRODUCTION

Logical design of a relational database, as indicated in [L7], consists of three phases :

1. **Conceptual schema design**, in which data requirements are represented in terms of data elements of a semantic data model. In this chapter, the RM/T semantic data model [L3,L5] is selected for this purpose.

2. **DBMS-independent logical design with the emphasis on integrity constraints**, in which the conceptual schema obtained from phase 1 is represented in terms of a set of DBMS-independent DDL statements which are able to fully express integrity constraints in the database. The DBMS-independent DDL in [L4,L5] is selected for this

purpose.

3. DBMS-dependent logical database design, in which the DBMS-independent DDL statements obtained from phase 2 are translated into the DDL statements of a particular relational DBMS. The integrity constraints are enforced by using the available integrity features of the DBMS or writing, if necessary, the integrity enforcement codes. The INGRES relational DBMS [L8,L11] is selected for this purpose.

2.2. CONCEPTUAL SCHEMA DESIGN

The RM/T [L3,L5] models the real world in term of entities. An entity is any distinguishable, concrete or abtract object. Entities with common properties are grouped into an entity type. There are three kinds of entity in RM/T: characteristics, associations, and kernels. A characteristic is an entity which qualifies and is existence-dependent on some other entity; the relationship between a characteristic and the entity it qualifies is m-to-1. An association is an m-to-n relationship between two independent entities (not necessarily distinct or of distinct types); "independent" means that they are not existence-dependent on each other. A kernel is an entity which is neither a characteristic nor an association; it is existence-independent of any other entity. An m-to-1

relationship between two independent entities a and b (not neccesarily distinct or of distinct types) is called a **designation** in RM/T; it is said that "a designates b" or "a is designative". In RM/T a designation is not considered as an entity, but rather as a property (discussed below). Kernels, characteristics, and associations are all allowed to be designative .

A property is a piece of infomation which describes an entity in some way. It may be immediate, nonimmediate, single-valued, or multivalued .

Entity type Y is a **subtype** of entity type X (or X is a **supertype** of Y) if every entity instance of Y is necessarily an instance of X. All properties (characterictics, associations, and designations) of X are also the ones of Y; this is called **property inheritance rule**; Y may have additional properties of its own. The main reason for introducing the concept of supertype (or subtype) is to eliminate the need for "property inapplicable" null values.

A given subtype may have multiple independent supertypes, in which case every instance of the subtype is simultaneously an instance of each of the supertypes. The subtype now inherits all properties (characteristics, associations, and designations) of all of its supertypes,

in addition to having properties of its own.

Now the data requirements must be analyzed to identify entities and classify them into kernels, associations, and charateristics. All the designations must also be identified. Because the data requirements are usually expressed in English, the rules presented in [L2] which converts English sentences to Entity-Relationship diagrams will be modified to apply them to the RM/T model :

1. A common noun is represented by a kernel entity type in RM/T .

2. A transitive verb corresponds to a m-to-n, m-to-1, or 1-to-1 relationship between two entity types :

a. If it is m-to-n, then the verb is represented by an associative entity type in RM/T.

b. If it is m-to-1 or 1-to-1 and the two entity types are independent, then the verb is represented by a designation in RM/T.

c. If it is m-to-1 or 1-to-1 and one entity type is existence-dependent on the other, then the former is a characteristic of the latter .

Note that the type of a relationship (i.e. m-to-n, m-to-1, or 1-to-1) and the existence-dependency can be determined only by the database designer because it is semantics-related.

3. An adjective is represented by a property of a kernel entity type in RM/T.

4. An adverb is represented by a property of an associative entity type, of a designative kernel entity type, or of a characteristic entity type, depending on whether the qualified verb corresponds to cases a, b, c, respectively, presented in rule 2.

5. For an English sentence of the form "the X of Y is Z", X may be treated as either :

a. a relationship set between Y and Z if Z is a proper noun (e.g. the father of Peter is Smith); then rule 2 can be applied to determine whether it is represented by an association, a designation, or a characteristic.

b. a property of Y if Z is not a proper noun (e.g. the color of the car is red).

6. A gerund is represented by an associative entity type, a designation , or a characteristic entity type, if the relationship represented by the gerund corresponds to cases a, b, or c, respectively, presented in rule 2.

7. A clause is represented by a high-level entity type which interconnects low-level entity types in RM/T.

8. An English sentence of the form "there are ... X in Y" is equivalent to "Y has...X".

As indicated in [L2], the above rules are not exhaustive; more rules need to be developed. Furthermore, those rules should be treated as guidelines only.

2.3. DBMS-INDEPENDENT LOGICAL DESIGN WITH EMPHASIS ON INTEGRITY CONSTRAINTS

Integity is concerned with the accuracy or correctness of data in the database. Every relational database must follow the two general integrity rules [L4,L6] :

1. Entity integrity, stating that primary key values must not be null.

2. Referential integrity, stating that the foreign key values must either match primary key values or be wholly null.

In addition, the database may have additional integrity rules of its own.

In [L4,L5], Date has proposed a general language for expressing any integrity rules. The language uses the cursor concept . A cursor is associated with only one relation and points to a specific tuple of that relation at any given time. Each integrity rule consists of three parts:

- A set of trigger conditions, each of which specifies when the checking of the constraint is to be done.

- The constraint itself.

- The violation-response, which specifies the action to be taken if the constraint is not satisfied.

Date in [L4,L5] has also proposed a special DBMS-independent Data Definition Language (DDL) which allows to express the entity and referential integrity rules within DDL statements. Note that existing relational DBMSs are rather weak in the support of data integrity; however, as indicated in [L6], the specification of integrity constraints may account for as much as 80% of a database definition. Thus the need for such a DBMS-independent DDL is very crucial to the database design process.

The syntax of a DBMS-independent DDL statement is as follows:

```
create table relation-name
 fields (field-name [, field-name])
 primary key (primary-key)
[foreign key (foreign-key identifies target
              null [not] allowed
              delete of target effect
              update of target-primary-key effect)]
```

where :

- "target" is a table-name;

- "target-primary-key" is the primary key of the target;

- "effect" is either **cascades** meaning that matching tuples in the referencing relation are also deleted or updated, or **restricted** meaning that the delete or update would fail if there exists any matching tuple, or **nullifies** meaning that the referential atttribute in the matching tuples are updated to null.

- **nulls allowed** means that the referential attribute is alllowed to take null values; **nulls not allowed** means not.

Note that the deletion of a tuple in the referencing relation has no effect on the referenced relations. However, the insertion of a tuple into the referencing relation or the update of a referential attribute in such a tuple require that there exists a tuple in either exactly one, at least one, or every one of the referenced relations (depending on the designer's decision) such that the corresponding primary key value matches the referetial attribute value in that inserted or updated tuple. The implementation of the entity and referential integrity rules will be discussed in section 2.3.1.

Now all the entities (i.e. kernels, associations, and characteristics), their properties, and all the designations indentified in the conceptual schema design must be represented by DBMS-independent DDL statements as follows [L7] :

1. Each entity type (kernel, associative, or characteristic) is represented by a DBMS-independent DDL statement.

- If it is kernel, then its primary key must be specified using primary key clause.

- If it is associative, each participating foreign key and its associated referential constraints must be specified using the foreign key clause. The primary key, which may be the combination of all those foreign keys,

must also be specified using the primary key clause.

If it is characteristic, the foreign key must have the following associated referential constraints which reflect the existence-dependency : nulls not allowed, delete of ... cascades, update of ... cascades. The primary key of the characteristic may be the combination of the foreign key and a property which can identify the characteristic within the given entity qualified by the characteristic.

2. Each designation is represented by a foreign key in the designating entity type. Specify the associated referential constraints of that foreign key using the foreign key clause.

Note that the specification of referential constraints in associations and designations must be based on the data semantics and the policies of the enterprise .

3. Each property p of the entity type A is represented as follows :

a. If p is immediate and single-valued, then it is represented as a field in A.

b. If p is immediate and multivalued, then it is represented by a characteristic entity type of A.

c. If p is nonimmediate and single-valued, then by definition there exists another entity type B such that A

designates B and p is an immediate and single-valued property of B; thus rule (a) can now be applied. For example, work-location is a nonimmediate and single-valued property of entity type Employee because it is an immediate and single-valued property of entity type Department which Employee designates.

d. If p is nonimmediate and multivalued, then by definition there exists another entity type B such that B is a characteristic of A and p is an immediate and single-valued property of B; thus rule (a) can now be applied. For example, course-locations is a nonimmediate and multivalued property of entity type Course because it is an immediate and singled-valued property of entity type Course Sections which is a characteristic of Course.

2.3.1 Implementation of Entity and Referential Integrity Rules

1. Entity integrity rule.

If p is the primary key of relation R, then this fact can be implemented as [L4,L5] :

```
change = insert of R from record r or update of
         R.p from r.p, where r is some record
         with the same fields as R;
if change is about to be executed
```

```
        then if (exist (R where R.p = r.p) or is-null
                (r.p))
             then execute violation-response;
```

Now if violation-response is executed and includes the reject operation, then change is not executed; otherwise, change is executed on the return from the application of the rule.

2. Referential Integrity Rule

If p is a foreign key of relation S which references relation R, then this fact can be implemented as :

```
case 1  /* insert into S or update S.p */
        change = insert of S from record s or update of S.p
                 from s.p, where s is some record
                 with the same fields as S;
        execute change ;
        if (not exists (R where R.p =S.p))
        then execute violation-response;
```

If violation-response is executed and includes the reject operation, then change is undone at that point.

```
case 2  /* delete of R or update of R.p */
        change = delete of R or update of R.p from r.p;
```

if change is about to be executed
then if (exist (S where S.p = R.p))
then violation-response

The violation-response may be cascades or nullifies, in which case the reject operation must not be executed, and thus allowing the change to be executed. If it is restricted, then the reject operation must be executed.

At the end of this phase, each DBMS-independent relation consists of one or more immediate, single-valued fields, one of which is the primary key. Thus it is already in the first normal form. Further normalization needs to be made to prevent update anomalies and data inconsistencies; these problems are caused by data redundancy [L9]. Normalization is discussed next.

2.3.2 Normalization

Let R be a DBMS-independent relation, X and Y be two fields of R. X is a **determinant** of Y if, for any instance of R, the same value of X is always associated with the same value of Y. All determinants in R can be shown by a **determinancy diagram**, in which an arrow from X to Y indicates that X is a determinant of Y. A **candidate key** of R is a field or a combination of fields which satisfies two time-independent properties : uniqueness and

minimality[L6].

Now a relation R is said to be in Boyce/Codd normal form if every determinant of R must be a candidate key [L9,L10]. Any relation which is in Boyce/Codd normal form is said to be well-normalized; otherwise, it is called badly-normalized. A practical method for transforming a badly-normalized relation into a set of well-normalized relations is as follows [L9,L10]: first, for each determinant which is not a candidate key, create a new relation which contains that determinant and its directly dependent (i.e. not transitively dependent) fields. Clearly, this new relation is well-normalized. Now, the candidate keys and their directly dependent fields left in the old relation becomes well-normalized, too. This method can be facilitated by using the determinancy diagram.

A relation is said to be fully-normalized if it contains no redundant data (see [L9] for a discussion about data redundancy). A well-normalized relation is usually (but not always) fully-normalized. A class of well-normalized relation which are not fully-normalized is identified as follows [L9]:

First let R be a DBMS-independent relation, X, Y and Z be all the fields of R. X multidetermines Y, written X ->> Y, if for a given X-value, all the possible pairs

(Y-value,Z-value) must exist in R where Y-value and Z-value belong to the set of Y-values and the set of Z-values corresponding to the given X-value, respectively. Since the definition is symmetric, we have : X ->> Y => X ->> Z

Y and Z are said to be independent attributes in R. Clearly, R(X,Y,Z) is well-normalized because (X,Y,Z) is the only determinant in R. However, R contains redundant data (an example can be found in [L9]). Now if R is decomposed into $R_1(X,Y)$ and $R_2(X,Z)$, then R_1 and R_2 are fully-normalized. This decomposition is lossless due to the fact X ->> Y/Z [L5].

A relation is said to be in fourth normal form if it contains no two independent fields. For example, R1 and R2 above are in fourth normal form.

After fully normalizing all the DBMS-independent relations, the integrity constraints specified in the old relations are modified accordingly to correspond to the new fully-normalized relations. This modification should be trivial .

Normalization tries to optimize update operations at the expense of retrieval operations [L7]. Now if two fields are very frequently retrieved together, very infrequently updated together, and separated by full normalization, then

it would be more beneficial to bring them back together again, i.e. to denormalize them [L7].

2.4. INGRES-DEPENDENT LOGICAL DESIGN

In this phase all the DBMS-independent relations and their associated entity and referential integrity constraints obtained from phase 2 are mapped into a particular relational DBMS. In INGRES system, each DBMS-independent relation is mapped into a create statement of the form [L8]:

```
create relation-name
        (col-def [, col-def])
        [with journaling]
```

where col-def is of the form :

```
col-name = data-type [null-spec]
```

where null-spec is one of the following : not null, with null, and not null with default (meaning that the field does not accept null values; however, if no value is specified for the field, it is automatically set to zero, blanks, or empty depending on the data type of the field).

2.4.1 Enforcement of Entity Integrity Rule in INGRES

The fact that p is the primary key of relation R can be enforced in INGRES as follows :

- Specify not null option for p in the create statement which corresponds to R. This will not allow p to accept null values.

- Specify either one of the following :

```
modify R to [c]isam unique on p

modify R to [c]hash unique on p

modify R to [c]btree unique on p
```

Each of these statement will guarantee the uniqueness of p within R at all time.

2.4.2 Enforcement of Referential Integrity Rule in INGRES

INGRES currently does not support referential integrity constraints. The following measures must be followed to ensure the integrity of the database :

a. Specify not null or with null, respectively, if

nulls not allowed or nulls allowed is specified for a foreign key.

b. Prohibit the following interactive QUEL operations:

- delete the referenced relation
- replace the primary key of the referenced relation
- insert the referencing relation
- replace the foreign key of the referencing relation

The prohibition can be done by means of the define permit statement which has the general form :

```
define permit operation(s)
        on      table [(field [,field])]
        to      user
        [at     terminals]
        [from   time1 to time2]
        [on     day1 to day2]
        [where  predicate]
```

The reason for such a prohibition is that it is impossible for an interactive end-user to enforce such referential constraints. Of course, he can still insert a tuple into the referenced relation or delete a tuple from the

referencing relation without causing any integrity problem.

c. For each foreign key, write a program which maintains referential constraints associated with it, using the algorithms presented in section 2.3.1 (i.e. implementation of integrity constraints). An end-user who wishes to perform the operations prohibited in (b) must go through an interface program, which in turn invokes the appropriate foreign key maintenance programs.

d. Similarly, application programs must go through an interface similar to the one in (c), i.e. not via QUEL inteface, to perform the operations prohibited in (b).

It is hopeful that features (c) and (d) will be provided by future INGRES. These features have been provided by INFORMIX relational DBMS [L13].

e. Finally, to enforce an integrity predicate which involves a single range variable, use the define integrity statement which has the form :

define integrity range-variable is predicate

2.5. CONCLUSION

In this chapter, several aspects for logical design of

relational databases have been described : RM/T conceptual data model, a language for expressing integrity constraints in relational databases, implementation of integrity constraints, redundancy, normalization, and how to carry out desirable relational integrity constraints in INGRES. There may be other practical factors which can affect the design process, but the above aspects are intrinsic and common to relational logical database design.

Chapter 3 PHYSICAL DESIGN FOR RELATIONAL DATABASES

ABSTRACT

A survey of DATAID-1 research techniques for secondary index selection in relational databases is conducted. Based on this; a program has been written in C to help automate secondary index selection process (Appendix C). A summary of practical method for index selection from INFORMIX relational DBMS is also presented.

3.1. INTRODUCTION

In relational DBMSs a software component, called the optimizer, selects the best access path in terms of total CPU and I/O costs to execute a given query. One of the factors which influences the optimizer's selection of the best access path is the set of secondary indices which are built over the relations of a relational database. The aim of the physical design for a relational database is to select an appropriate set of secondary indices so that the optimizer can make the best possible decisions.

In this chapter a method for the secondary index selection problem will be presented. This method has been discussed in various papers, as part of the DATAID-1 project (from [P1] to [P7]). See Appendix C for the implementation in C of this method.

3.2. ASSUMPTIONS AND DEFINITIONS

In general, the following assumptions and definitions are applied to most relational DBMSs.

1. Indices are implemented as B⁺-trees. Each leaf node corresponds to one data page and contains an index value followed by a set of tuple identifiers (TIDs) of the tuples in which the index value appears. There are two kinds of indices: primary index which is built on the primary key of a relation, and secondary index which is built on other attributes. The intermediate level of the B⁺-tree are ignored in the physical database design problem.

2. A relation can be accessed sequentially or via only one of the indices (primary or secondary) built on that relation.

3. An update on attribute j made by query q cannot use the index built on j, but rather the primary index or a sequential scan. The reason is as follows [P3]: "When a column entry in a given tuple is modified, the TID associated to the tuple is removed from the group of TIDs following the old key value in the index on that column and is inserted into the group of the new key value. Access to the tuples through an index that is currently modified may

lead to the retrieval of the same TID more than once.". However, if appropriate concurrency control is provided, then this assumption can be removed.

4. Uniform distribution is assumed, i.e. values of a column are uniformly distributed between maximum and minimum values in that column. As a result of this assumption, the selectivity factor of a predicate p of a relation R, which is defined as the fraction of the tuples in R satisfying p, is computed as follows [P8]:

$$
f(p,R) = \begin{cases}
1) & (value-MINKEY)/(MAXKEY-MINKEY) \\
 & \text{if } p : a < value \\
2) & 1/NKEY \text{ if } p : a = value \\
3) & (MAXKEY-value)/(MAXKEY-MINKEY) \\
 & \text{if } p : a > value \\
4) & (value2-value1)/(MAXKEY-MINKEY) \\
 & \text{if } p : value1 \leq a \leq value2 \\
5) & card(V)/NKEY \text{ if } p \in \text{set of} \\
 & \text{values } V
\end{cases}
$$

where MINKEY, MAXKEY, NKEY are the minimum value, maximum value, the number of different values, respectively, in the attribute a.

Assuming that uniform distribution stated above is in effect and p_1 and p_2 are predicates on two different and

independent columns, then for logical combination of predicates, the selectivity factors are computed as follows [P8]:

$f(p_1 \text{ AND } p_2,R) = f(p_1,R)*f(p_2,R)$

$f(p_1 \text{ OR } p_2,R) = f(p_1,R)+f(p_2,R)-f(p_1,R)*f(p_2,R)$

$f(\text{NOT } p,R) = 1-f(p,R)$

5. Let $R(a_1.....a_n)$ be a relation; q be a query on R having the form:

```
range of r is R
retrieve ...
from R
where a_i = value
```

The cost of q in terms of the number of I/O operations (CPU cost is ignored because it is trivial when compared with I/O cost), using the index on a_i, is computed as follows:

- The expected number of tuples which satisfies the predicate, p: a_i = value is:

$$ET = f(a_i,p) * ntpl(R)$$

where ntpl(R) is the number of tuples in R.

Now the expected number of data pages to be accessed is computed, using the CARDENAS [P1] formula, as follows:

$$EP = npag(R)*(1-(1-1/npag(R))^{nT})$$

where npag(R) is the number of data pages in R.

- The expected number of index leaves to be accessed is:

$$EL = f(a_i,p) * nleaf(a_i)$$

where $nleaf(a_i)$ is the number of leaves of the index built on attribute a_i.

The expected execution cost of q in terms of the number of I/O operations is :

$$E = EP + EL$$

Joins are done by the nested loop method [P2]. For example, consider the join:

q: retrieve ...
from R_1,R_2
where p_1
and p_3

and p_2

where p_1, p_2 are predicates on R_1, R_2 respectively; p_3 is the join predicate.

According to the nested loop method, q is decomposed into two single-relation queries:

q_1: retrieve ...
from R_1
where p_1

q_2: retrieve ...
from R_2
where p_2
and p_3

Then the expected execution cost for q is:

$$E = E_1 + ntpl(R_1,p_1)*E_2$$

where E_1, E_2 are the execution costs for q_1, q_2 respectively; $nptl(R_1,p_1)$ is the number of tuples in R_1 which satisfies p_1

The weight of a query is defined as a function which depends on a number of factors such as the frequency,

criticality of response times, etc. Now if w is the weight of q then the weights of q_1 and q_2 are $w_1 = w$ and $w_2 = ntpl(R_1,p_1)*w$, respectively.

3.3. DESIGN PROCEDURE

The design procedure consists of the following steps:

1. All the joins in the workload are decomposed into single-relation queries. The set of attributes referenced in the single-relation queries becomes the set of candidate secondary indices to be selected. The primary indices associated with primary keys are assumed to already exist.

2. For each relation R_h (h = 1,..,NR), where NR is the number of relations in the database, $ISET_h$ and Q_h denote the sets of candidate secondary indices and queries operating on R_h, respectively. Build the following matrices:

- $A^h(i,j)(i \in Q_h, j \in ISET_h)$. Each element $A^h(i,j)$ represents the execution cost of query q_i when only the index on attribute j is available (in addition to the primary index and the sequential scan). In general, $A^h(i,j)$ is computed using the assumption (5) in section 3.2. If q_i is an update query, then $A^h(i,j)$ must also include the update cost of R_h and possibly the update cost of the

primary index.

- $U^h(i,j)(i \in Q_h, j \in ISET_h)$. Each element $U^h(i,j)$ represents the update cost of the index on attribute j due to query q_i.

- $M^h(j)(j \in ISET_h)$. Each element $M^h(j)$ represents the number of memory pages required by the index on attribute j.

- $P^h(i)(i \in Q_h)$. Each element $P^h(i)$ represents the minimum execution cost of query q_i between using the primary index and using the sequential scan.

The computation of A^h, U^h, P^h, and M^h are system-dependent and must follow the same method used by the optimizer of a particular DBMS.

Having built A^h, U^h, P^h, and $M^h(h=1,..,NR)$, let:

$$ISET = \bigcup_{h=1}^{NR} ISET_h;$$

$$Q = \bigcup_{h=1}^{NR} Q_h$$

Then build the following global matrices GA, GU, P, and M:

$$GA_{ij} = A^h(i,j),\ I_j \in ISET_h$$

$$P^h(i),\ I_j \notin ISET_h$$

$$GU_{ij} = \begin{cases} U^h(i,j), & I_j \in ISET_h \\ 0, & I_j \notin ISET_h \end{cases}$$

$$P_i = P^h(i),\ q_i \in Q_h$$

$$M_j = M^h(j),\ I_j \in ISET_h$$

3. For each index $I_j \in ISET$, compute the global cost G_j as follows:

$$G_j = \sum_{q_i \in Q} (GA_{ij} + GU_{ij}) * w_i$$

The meaning of G_j is that it is the total execution cost of all queries when only the index I_j (in addition to the sequential scan and primary indices of relations in the database) is present.

Now order all the indices I_j in ISET according to the increasing or decreasing values of G_j.

4. First, the concept of dominance is introduced. Let BSET $\subseteq$ ISET. For each $q_i \in Q$, compute:

$$B_i = \min \{ GA_{ij} \mid I_j \in BSET \} + \sum_{I_j \in BSET} U_{ij}$$

The first component of B_i is the best execution cost of q_i in BSET, the second one is the total update cost of all indices in BSET due to q_i.

Let GM denotes the maximum number of data pages which can be available for all indices in the database; BSET and BSET' be two sets of secondary indices. BSET is said to be dominated by BSET' if :

$$\sum_{q_i \in Q} w_i * B_i' \quad < \quad \sum_{q_i \in Q} w_i * B_i$$

$$\sum_{I_j \in BSET'} M(j) \quad < \quad GM$$

Now the algorithm for producing an efficient set of secondary indices is summarized as follows:

1. Initiate BSET with the primary indices of all relations.

2. A single pass over the set of secondary indices in ISET in the order obtained from (3) above will produce a suboptimal solution [P1]. At each step, a secondary index is added to BSET. If the new BSET dominates the previous one, then the index is retained; otherwise, it is eliminated. In case it is retained, some secondary indices in BSET may no longer be the best access path for <u>any</u> query in Q; these indices must be removed from the BSET. An

example of this method can be found in [P6].

3.4. PRACTICAL METHOD FOR INDEX SELECTION

In general, indices can speed up some database operations such as data searching and sorting. However, they can also slow down others such as data insertion and modification because both data and indices must be updated. Indices also take up disk memory. Following are the practical guidelines for index selection [P7] :

1. Create an index on any attribute participating in a join operation.

2. Create an index on any attribute used to search or sort data. If data are to be sorted on multiple attibutes, create a composite index on those attributes. For example, if data are to be sorted on first name within last name, then an index on (last name,first name) should be created.

3. Do not create an index on any attribute which has a large number of duplicate values. For example, do not index an attribute that contains either Y or N.

4. Use Standard SQL statement create unique index to prevent duplicate entries in the indexed attribute. For example, duplicate social insurance numbers should not be

allowed.

3.5. INGRES FACILITIES FOR INDICES

An index can be created in INGRES by an index statement which has the form:

index on rel-name is index-name(attribute [, attribute])

Once an index is created, it is stored in an operating system file and automatically maintained by INGRES.

An index can be destroyed by a destroy command which has the form:

destroy index-name

It is not possible to build an index over an index, nor over on a system table.

3.6. CONCLUSION

Index selection is the most important problem that must be considered when performing physical design for relational databases. All other physical aspects, such as choice of best access paths are performed automatically by the

optimizer, and thus are concealed from the designer. With a good choice of indices over the relations, the designer can achieve both good performance as in network or hierarchical DBMSs and the simplicity advantage of relational DBMS.

REFERENCES

Design of Distributed Databases

[D1] R.J. Ballard, "INGRES/Distributed Database - Meeting Business Needs," in Relational Databases : State of the Art Report, Infotech, 1986.

[D2] C. Batini and M. Lenzerini. "A Computer-Aided Methodology for Conceptual Database design." Information Systems, vol.7, No.3, 1982.

[D3] C. Batini, M. Lenzerini, and M. Moscarini. "Views Integration. " in Methodology and Tools for Database Design. S. Ceri, ed., North-Holland, 1983.

[D4] C. Batini and M. Lenzerini. "A Methodology for Data Schema Integration in the Entity-Relationship Model." IEEE Transactions on Software Engineering, Vol.SE.10, No.6, 1984.

[D5] C. Batini and S. Ceri. "Database Design : Methodologies, Tools, and Environments." Proc. ACM-SIGMOD, 1985.

[D6] Y. Breitbart, P. Olson, and G. Thompson. "Database

Integration in a Distributed Heterogeneous Database System." Proc. International Conference on Data Engineering, 1986.

[D7] S. Ceri, M. Negri, and G. Pelagatti. "Horizontal Partitioning in Database Design." Proc. SIGMOD Conference, 1982.

[D8] S. Ceri and S. Navathe. "A Methodology for the Distribution Design of Databases." Proc. Compcon83, San Francisco, 1983.

[D9] S. Ceri, S. Navathe, and G. Wiederhold. "Distribution Design of Logical Database Schemas." IEEE Transactions on Software Engineering, Vol. SE-9, No.4, 1983.

[D10] S. Ceri and G. Pelagatti. Distributed Databases : Principles and Systems, McGraw-Hill, 1984.

[D11] S. Ceri, B. Percini, and G. Wiederhold. "An overview of Research in the Design of Distributed Databases." Proc. Database Engineering.

[D12] S. Ceri and B. Percini. "DATAID-D : Methodology for Distributed Database Design." in Computer-Aided Database Design : The DATAID Project, A. Albano et al., ed.,

North-Holland, 1985.

[D13] C.J. Date. A Guide to INGRES, Addison-Wesley, 1987.

[D14] U, Dayal and H-W Hwang. "View Definition and Generalization for Database Integration in a Multibase System." IEEE Transactions on Software Engineering, Vol. SE-14, No.6, 1984.

[D15] W. Effelsberg and M. Mannimo. "Attribute Equivalence in Global Schema Design for Heterogeneous Distributed Databases." Information Systems, Vol.9, NO.3/4, 1984.

[D16] R. Elmari and G. Wiederhold. "Properties of Relationships and their Representation." Proc. National Computer Conference, AFIPS, Vol.49, 1980.

[D17] R. Elmari, J. Larson, S. Navathe, and T. Sashidhar. "Tools for View Integration." Proc. Database Engineering, 1985.

[D18] R. Elmari and S. Navathe. "Object Integration in Logical Database Design." Proc. IEEE COMPDEC Conference, 1984.

[D19] J.B. Grimson. "Distributed Relational Database Systems." in Relational Databases : State of the Art

Report, Infotech, 1986.

[D20] J. Kalash, L. Rodgin, Z. Fong, and J. Anton. INGRES Reference Manual, Version 8.

[D21] M. Mannimo and W. Effelsberg. "Matching Techniques in Global Schema Design." Proc. IEEE COMPDEC Conference, 1984.

[D22] S. Navathe and S. Gadgil. "A Methodology for View Integration in Logical Database Design." Proc. 8th VLDB Conference, 1982.

[D23] S. Navathe, T. Sashidhar, and R. Elmari. "Relationship Merging in Schema Integration." Proc. 10th VLDB Conference, 1984.

[D24] S. Navathe, S. Ceri, G. Wiederhold, and J. Dou. "Vertical Partitioning Algorithms for Database Design." ACM-TODS, Vol.9, No.4, 1984.

[D25] D. Sacca and G. Wiederhold. "Database Partitioning in a Cluster of Processors." ACM-TODS, Vol.10, No.1, 1985.

[D26] J.M. Smith, P.A. Bernstein, U. Dayal, n. Goodman, T. Landers, K.W.T. Lin, and E. Wong. "MULTIBASE - Integrating Heterogeneous Distributed Satabase Systems." Proc. National

Computer Conference, AFIPS, 1981.

[D27] W. Staniszkis. "Integrating Heterogeneous Databases." in Relational Databases : State of the Art Report, Infotech, 1986.

[D28] M. Stonebraker, P. Kreps, E. Wong, and G. Held. "The Design and Implementation of INGRES." ACM-TODS, Vol.1, No.3, 1976.

[D29] M. Stonebraker. "The Design and Implementation of Distributed INGRES." in The INGRES Papers : Anatomy of a Relational Database System, Addison-Wesley, 1986.

[D30] D.R. Howe. Data Analysis for Database Design, Edward Arnold, 1983.

Logical Design for Relational Databases

[L1] P.P. Chen. " The Entity-Relationship Model : Toward a Unified View of Data." ACM TODS, Vol.1, No.1, 1976.

[L2] P.P. Chen. "English Sentences Structure and Entity-Relationship Diagrams." Information Sciences, Vol.29, 1983.

[L3] E.F. Codd. "Extending the Database Relational Model to Capture More Meanig." ACM TODS, Vol.4, No.3, 1979.

[L4] C.J. Date. "Referential Integrity." Proc.7th International Conference on Very Large Data Bases, 1981.

[L5] C.J. Date. An Introduction to Database Systems, Volume 2, Chapter 2 and 6, Addition-Wesley, 1986.

[L6] C.J. Date. An Introduction to Database Systems, Vol.1, 4th Edition, Chapters 12, 15, 19, and 25, Addison-Wesley, 1986.

[L7] C.J. Date. "A Practical Approach to Database Design." Relational Database : Selected Writings, C.J. Date, Addition-Wesley, 1986.

[L8] C.J. Date. A Guide to INGRES, Addition-Wesley, 1987.

[L9] D.R. Howe. "Duplication : Is It Redundant?." Computer Bulletin, 1982.

[L10] D.R. Howe. Data Analysis for Database Design, Edward Arnold, 1983.

[L11] J.Kalash, L.Rodgin, Z.Fong, and J.Anton. INGRES

Reference Manual, Version 8.

[L12] W.Kent. "A Simple Guide to Five Normal Forms in Relational Database Theory." Communications of the ACM, Vol.26, No.2, 1983.

[L13] INFORMIX Reference Manual, Relational Database Systems Inc., Version 2.00.00, 1986.

Physical Design for Relational Databases

[P1] F. Bonfatti, D. Maio, and P. Tiberio, "A Seperability - Based Method for Secondary Selection in Physical Database Design," in Methodology and Tools for Database Design, S. Ceri ed., Noth. Holland, 1983.

[P2] R. Bonanno, D. Maio, and P. Tiberio, "An Approximation Algorithm for Secondary Index Selection in Relational Database Physical Design," The Computer Journal, Vol. 28, No. 4, 1985.

[P3] M. Schkolnick and P. Tiberio, "Estimating the Cost of Updates in a Relational Database," ACM-TODS, Vol. 10, No. 2, June 1985

[P4] D. Miao, M. Scalas, and P. tiberio, "On Estimating Access Costs in Relational Database," Information Processing Letters, Vol. 18, No. 3, October 1984.

[P5] V. Antonellis and A. Lera, "DATAID-1: A Database Design Methodology," Information Systems, Vol. 10, No. 2, 1985.

[P6] V. Antonllis and A. Lera, "A Case Study of Database Design Using the DATAID Approach," Information Systems, Vol. 10, No. 3, 1985.

[P7] INFORMIX Reference Manual, Relational Database Systems Inc., Version 2.00.00, 1986.

[P8] D. Maio, C.Sartori, and M.R. Scalas, "Architecture of a Physical Design Tool for Relational DBMSs," in Computer-Aided Database Design, North-Holland, 1985.

APPENDIX A

Program Listing for HAD Method 1 and NAVF Method 2

```
/*************************************************************************/
/* PURPOSE : This interactive program find the allocation sites for      */
/*           a set of fragments. Both nonreplicated and replicated       */
/*           cases are considered. Then it will ask if we wish to        */
/*           vertically partition any fragment. When the two vertical    */
/*           fragments are specified, it will find the allocation sites  */
/*           for them. This process is repeated until we do not wish     */
/*           to vertically partition any more fragment                   */
/* INPUT : . Update-retrieval access ratio                               */
/*         . Transaction frequency matrix                                */
/*         . Transaction retrieval access matrix                         */
/*         . Transaction update access matrix                            */
/*         . Two vertical fragments for each of the given fragment       */
/*           we want to vertically partition                             */
/* OUTPUT : . The nonreplicated and replicated solutions for the         */
/*            given set of fragments                                     */
/*          . The allocation sites for each of the given fragments       */
/*            we wish to vertically partition                            */
/* METHOD : HAD Method 1 and NAVF Method 2, Chapter 1, Major Report      */
/*************************************************************************/
#define MAXI 50   /* max. no. of fragments */
#define MAXJ 10   /* max. no. of sites */
#define MAXK 20   /* max. no. of operations */
#define MAXL 15   /* max. no  of attributes per fragments */
#include <stdio.h>
int f[MAXK][MAXJ];
int r[MAXK][MAXI];
int u[MAXK][MAXI];
struct el {
        int no;
        struct el *next;
};
/*-----------------------------------------------------------------------*/
main()
{
        void append();
        int use(), test(), bntcomp();
        char ch;
        int c, nfrag, nsite, nop;
        int i, j, k, l;
        int pmin, cmin, low;
        int sv, ct, flag, jj;
        int frno, nattr, attrno, fvfr, svfr;
        int bnt, pbnt, loc1, loc2;
        int s, t;
        int site[MAXJ];
        int au[MAXK][MAXL];
        struct el *ptr;
        struct el *h1, *h2, *hop, *os, *ot, *e1, *e2, *e3;
        printf("          INTERACTIVE PROGRAM TO SOLVE THE NONREPLICATED AND\n");
        printf("          REPLICATED HORIZONTAL ALLOCATION PROBLEM, AND THE\n");
```

```
52      printf("          NONREPLICATED VERTICAL ALLOCATION PROBLEM\n");
53      printf("\n\n\n\n");
54      printf("          METHOD : HAD METHOD 1 AND NAVF METHOD 2, CHAPTER 1.\n");
55      printf("                   MAJOR REPORT\n");
56      printf("\n\n\n\n");
57      printf("Program starts now . . . .\n\n\n");
58      /* read update-retrieval access ratio */
59      printf("Enter the update-retrieval access ratio\n");
60      scanf("%d", &c);
61      /* read no. of fragments, no. of sites, no. of operations */
62      printf("Enter no.of fragments, no. of sites, no. of operations\n");
63      scanf("%d%d%d", &nfrag, &nsite, &nop);
64      /* read frequency matrix */
65      printf("Enter the frequency matrix\n");
66      for (k=0; k<nop; k++)
67              for (j=0; j<nsite; j++)
68                      scanf("%d", &f[k][j]);
69      /* read retrieval matrix */
70      printf("Enter the retrieval matrix\n");
71      for (k=0; k<nop; k++)
72              for (i=0; i<nfrag; i++)
73                      scanf("%d", &r[k][i]);
74      /* read update matrix */
75      printf("Enter the update matrix\n");
76      for (k=0; k<nop; k++)
77              for (i=0; i<nfrag; i++)
78                      scanf("%d", &u[k][i]);
79      ch = getchar();
80
81      /* nonreplicated horizontal partitioning */
82      printf("****** NONREPLICATED HORIZONTAL ALLOCATION SOLUTION ******\n\n");
83      for (i=0; i<nfrag; i++) {
84              pmin = 0;
85              for (j=0; j<nsite; j++) {
86                      cmin = 0;
87                      for (k=0; k<nop; k++)
88                              cmin = cmin + f[k][j]*(r[k][i] + c*u[k][i]);
89                      if (cmin > pmin) {
90                              pmin = cmin;
91                              low = j;
92                      }
93              }
94              site[i] = low;
95              printf("Fragment %d should be allocated to site %d\n", i+1, site[i]+1);
96      }
97
98      /* replicated horizontal partitioning */
99      printf("\n\n");
100     printf("****** REPLICATED HORIZONTAL ALLOCATION SOLUTION ******\n\n");
101     for (i=0; i<nfrag; i++) {
102             flag = 0;
```

```
103                 for (j=0; j<nsite; j++){
104                         sv = ct = 0;
105                         for (k=0; k<nop; k++)
106                                 sv = sv + f[k][j]*r[k][i];
107                         for (jj=0; jj<nsite; jj++)
108                                 if (jj != j)
109                                         for (k=0; k<nop; k++)
110                                                 ct = ct + f[k][jj]*c*u[k][i];
111                         if (sv-ct > 0) {
112                                 printf("Fragment %d should be replicatedly assign to site %d\n", i+1, j+1);
113                                 flag = 1;
114                         }
115                 }
116                 if (!flag) {
117                         printf("There is no benefit to replicatedly assign fragment %d\n", i+1);
118                         printf("to any other site\n");
119                 }
120         }
121         /* vertical partitioning */
122         printf("\n\n");
123         printf("******.NONREPLICATED VERTICAL ALLOCATION SOLUTION ******\n\n");
124         printf("Do you want to vertically partition any fragment?\n");
125 LAB:
126         while (ch != 'y' && ch != 'n')
127                 ch = getchar();
128         if (ch == 'y') {
129
130                 printf("Which fragment do you want to vertically partition?\n");
131                 scanf("%d", &frno);
132                 h1 = NULL;
133                 printf("How many attributes are there in fragment %d\n", frno);
134                 scanf("%d", &nattr);
135                 printf("Indicate the attributes of the first vertical fragment\n");
136                 ch = ' ';
137                 while (ch != '\n') {
138                         scanf("%d%c", &attrno, &ch);
139                         append(&h1, attrno-1);
140                 }
141                 printf("Indicate the attributes of the second vertical fragment\n");
142                 h2 = NULL;
143                 ch = ' ';
144                 while (ch != '\n') {
145                         scanf("%d%c", &attrno, &ch);
146                         append(&h2, attrno-1);
147                 }
148                 printf("Enter the attribute usage matrix for fragment %d\n", frno);
149                 for (k=0; k<nop; k++)
150                         for (l=0; l<nattr; l++)
151                                 scanf("%d", &au[k][l]);
152                 /* hop is pointer to list of operations referencing fragment frno */
153                 hop = NULL;
```

```
154         for (k=0; k<nop; k++)
155                 if (r[k][frno-1] != 0 || u[k][frno-1] != 0)
156                         append(&hop, k);
157         /*find sites s and t for vertical fragments h1 and h2 that result in */
158         /*best benefits */
159         flag = 0;
160         pbnt = 0;
161         for (s=0; s<nsite; s++)
162                 for (t=0; t<nsite; t++)
163                         if (s != t && t != site[frno-1] && s != site[frno-1]) {
164                                 ptr = hop;
165                                 os = ot = o1 = o2 = o3 = NULL;
166                                 while (ptr != NULL) {
167                                         /* build list os */
168                                         if (f[ptr->no][s] != 0)
169                                                 if (use(ptr->no, h1, au) && !use(ptr->no, h2, au))
170                                                         append(&os, ptr->no);
171                                         /* build list *ot */
172                                         if (f[ptr->no][t] != 0)
173                                                 if (use(ptr->no, h2, au) && !use(ptr->no, h1, au))
174                                                         append(&ot, ptr->no);
175                                         /* build list *o1 */
176                                         if (f[ptr->no][site[frno-1]] != 0) {
177                                                 if (use(ptr->no, h1, au) && !use(ptr->no, h2, au))
178                                                         append(&o1, ptr->no);
179                                                 if (use(ptr->no, h2, au) && !use(ptr->no, h1, au))
180                                                         append(&o1, ptr->no);
181                                         }
182                                         /* build list *o2 */
183                                         if (f[ptr->no][site[frno-1]] != 0)
184                                                 if (use(ptr->no, h1, au) && use(ptr->no, h2, au))
185                                                         append(&o2, ptr->no);
186                                         /* build list *o3 */
187                                         if (test(ptr->no, site[frno-1], s, t, nsite))
188                                                 if (use(ptr->no, h1, au) && use(ptr->no, h2, au))
189                                                         append(&o3, ptr->no);
190                                         ptr = ptr->next;
191                                 } /* end while (ptr != NULL) */
192                                 bnt = bntcomp(os, ot, o1, o2, o3, site[frno-1], s, t, c, frno, nsite);
193                                 if (bnt > pbnt) {
194                                         pbnt = bnt;
195                                         flag = 1;
196                                         loc1 = s;
197                                         loc2 = t;
198                                 }
199                         } /* end if (s != t ...) */
200         if (flag) {
201                 printf("The first and second vertical fragments of fragment %d\n", frno);
202                 printf("should be allocated to sites %d and %d, respectively\n", loc1+1, loc2+1);
203         }
204         else {
```

```
                        printf("There is no benefit to vertically partition fragment %d\n",frno);
                        printf("and assign them to any other different sites\n");
                }
                printf("Do you want to vertically partition any other fragment\n");
                goto LAB;
        }
} /* end main() */
/*-------------------------------------------------------------------------*/

/* function to append number num to list pointed by *head */
void append(head, num)
struct el **head;
int num;
{
        struct el *p;
        p = (struct el *) malloc(sizeof(struct el));
        p->no = num;
        p->next = *head;
        *head = p;
} /* end append() */
/*-------------------------------------------------------------------------*/

/* function to determine if operation op uses any attributes in the list pointed by pt using matrix table */
int use(op, pt, table)
int op;
struct el *pt;
int table[][MAXL];
{
        struct el *p;
        int flg;
        p = pt;
        flg = 0;
        while (p != NULL && !flg) {
                if (table[op][p->no] != 0)
                        flg = 1;
                p = p->next;
        }
        return(flg);
} /* end use() */
/*-------------------------------------------------------------------------*/

int test(op, rr, ss, tt, nst)
int op, rr, ss, tt, nst;
{
        int j, flg;
        flg = 0;
        j = 0;
        while (!flg && j < nst) {
                if (j != rr && j != ss && j != tt)
                        if (f[op][j] != 0)
                                flg = 1;
```

```
256                     j++;
257             }
258             return(flg);
259     } /* end test() */
260     /*-------------------------------------------------------------------------*/
261
262     int bntcomp(bos, bot, boi, bo2, bo3, br, bs, bt, bc, bfrno, bnsite)
263     struct el *bos, *bot, *boi, *bo2, *bo3;
264     int br, bs, bt, bc, bfrno, bnsite;
265     {
266             struct el *p;
267             int savings, costs, j;
268             savings = costs = 0;
269             p = bos;
270             while (p != NULL) {
271                     savings = savings + f[p->no][bs]*(r[p->no][bfrno-1]+bc*u[p->no][bfrno-1]);
272                     p = p->next;
273             }
274             p = bot;
275             while (p != NULL) {
276                     savings = savings + f[p->no][bt]*(r[p->no][bfrno-1]+bc*u[p->no][bfrno-1]);
277                     p = p->next;
278             }
279             printf("** savings = %d\n", savings);
280             p = boi;
281             while (p != NULL) {
282                     costs = costs + f[p->no][br]*(r[p->no][bfrno-1]+bc*u[p->no][bfrno-1]);
283                     p = p->next;
284             }
285             p = bo2;
286             while (p != NULL) {
287                     costs = costs + 2*f[p->no][br]*(r[p->no][bfrno-1]+bc*u[p->no][bfrno-1]);
288                     p = p->next;
289             }
290             for (j=0; j<bnsite; j++)
291                     if (j != br && j != bs && j != bt) {
292                             p = bo3;
293                             while (p != NULL) {
294                                     costs = costs + f[p->no][j]*(r[p->no][bfrno-1]+bc*u[p->no][bfrno-1]);
295                                     p = p->next;
296                             }
297                     }
298             printf("** costs = %d\n\n", costs);
299             return(savings-costs);
300     } /* end bntcomp() */
301
```

First, this interactive program will ask you to enter :

- update-retrieval access ratio (note that an update access is usually more expensive than a retrieval access).
- number of fragments, number of sites, and number of operations.
- frequency matrix (i.e. number of activations for each operation at each site per unit time).
- retrieval access matrix (i.e. number of tuples retrieved for each fragment by each operation on each of its activations).
- update access matrix (i.e. number of tuples updated for each fragment by each operation on each of its activations).

Then the program will output the best allocation sites for the given fragments both in nonreplicated and replicated allocation cases.

Next, it will ask you whether to vertically partition any fragment. If the answer is yes, it will ask you for the fragment number, the number of attributes of that fragment, the attributes of the first and second vertical fragments, the attribute usage matrix of operations on that fragment. The program will then either find the best allocation sites for the two vertically partitioned fragments or it will tell you that there is no benefit to vertically partition that fragment. This process is repeated until you indicate that you no longer wish to vertically partition any more fragments.

APPENDIX B

Program Listing for NAVF Method 1

```
/*****************************************************************************/
/* PURPOSE : This program finds the best allocation sites for the two        */
/*           vertically partitioned fragments of a relation                  */
/* INPUT   : . Relation R and two vertically partitioned fragments L and U   */
/*           . Lengths of attributes of R                                    */
/*           . Transaction access vector                                     */
/*           . Transaction frequency matrix                                  */
/*           . Access cost per tuple, storage cost per byte, and transmission*/
/*             cost per byte                                                 */
/*           . Attribute usage matrix                                        */
/* OUTPUT  : The best allocation sites for the two vertically partitioned    */
/*           fragments                                                       */
/* METHOD  : NAVF Method 1, Chapter 1, Major Report (Non-replicated          */
/*           Allocation of Vertical Fragments)                               */
/*****************************************************************************/
#include <stdio.h>
#define MAXJ 10  /* max. no. sites */
#define MAXK 20  /* max. no. transactions */
#define MAXL 15  /* max  no. attributes of a relation */
#define MAXCOST 50000000
struct el {
        int no;
        struct el *next;
};
int f[MAXK][MAXJ], a[MAXK], leng[MAXL], au[MAXK][MAXL];
int nattr, fvfr, svfr, nsite, ntr, c1, c2, c3;
main()
{
        void append();
        int use(), access(), trans();
        int i, attrno, k, j, l, s, t, bs, bt, w1, w3, z, prez;
        struct el *h1, *h2, *lt, *ut, *it, *ptr;
        FILE *fp, *fopen();
        printf("          PROGRAM TO FIND THE BEST ALLOCATION SITES FOR THE\n");
        printf("          TWO VERTICALLY PARTITIONED FRAGMENTS OF A RELATION\n");
        printf("\n\n\n");
        printf("METHOD : NAVF Method 1, Chapter 1, Major Report\n");
        printf("(Non-replicated Allocation of Vertical Fragments)\n\n\n");
        fp = fopen("dfile3", "r");
        if (fp == NULL) {
                fprintf(stderr, "cannot read the file\n");
                exit(1);
        }
        printf("SAMPLE INPUT :\n\n");
        /* read in numbers of attributes of R, first vertical fragment, and */
        /* second vertical fragment */
        fscanf(fp, "%d%d%d", &nattr, &fvfr, &svfr);
        printf("        Attributes of relation R :\n");
        printf("          ");
        for (i = 0; i < nattr; i++)
                printf("%d    ", i+1);
```

```
52          printf("\n\n");
53          /* h1 points to the list of attributes of first vertical fragments */
54          printf("       : Attributes of the first vertical fragment L :\n");
55          printf("           ");
56          h1 = NULL;
57          for (i = 0; i < fvfr; i++) {
58                  fscanf(fp, "%d", &attrno);
59                  printf("%d    ", attrno);
60                  append(&h1, attrno-1);
61          }
62          printf("\n\n");
63          /* h2 points to the list of attributes of second vertical fragment */
64          printf("       . Attributes of the second vertical fragment U :\n");
65          printf("           ");
66          h2 = NULL;
67          for (i = 0; i < svfr; i++) {
68                  fscanf(fp, "%d", &attrno);
69                  printf("%d   ", attrno);
70                  append(&h2, attrno-1);
71          }
72          printf("\n\n");
73          /* read in the numbers of sites and transactions */
74          fscanf(fp, "%d%d", &nsite, &ntr);
75          /* read in access cost per tuple, storage cost per byte, and transmission */
76          /* cost per byte */
77          fscanf(fp, "%d%d%d", &c1, &c2, &c3);
78          printf("       . Access cost per tuple, storage cost per byte, and\n");
79          printf("         transmission cost per byte :\n");
80          printf("           %d    %d    %d\n", c1, c2, c3);
81          /* read in the lengths of attributes */
82          printf("       . Lengths of attributes of R :\n");
83          printf("           ");
84          for (l=0; l<nattr; l++) {
85                  fscanf(fp, "%d", &leng[l]);
86                  printf("%d    ", leng[l]);
87          }
88          printf("\n\n");
89          /* read in transaction access vector */
90          printf("       . Transaction access vector (each access retrieves\n");
91          printf("         or updates one tuple) :\n");
92          printf("           ");
93          for (k=0; k<ntr; k++) {
94                  fscanf(fp, "%d", &a[k]);
95                  printf("%d    ", a[k]);
96          }
97          printf("\n\n");
98          /* read in transaction frequency matrix */
99          printf("       . Transaction frequency matrix :\n");
100         printf("           ");
101         for (k=0; k<ntr; k++) {
102                 for (j=0; j<nsite; j++) {
```

```
                fscanf(fp, "%d", &f[k][j]);
                printf("%d    ", f[k][j]);
        }
        printf("\n          ");
    }
    printf("\n\n");
    /* read in attribute usage matrix */
    printf("        Attribute usage matrix :\n");
    printf("          ");
    for (k=0; k<ntr; k++) {
        for (l=0; l<nattr; l++) {
            fscanf(fp, "%d", &au[k][l]);
            printf("%d    ", au[k][l]);
        }
        printf("\n          ");
    }
    printf("\n\n");
    printf("\n\n\nOUTPUT :\n\n");
    /* lt : list of transactions accessing only first vertical fragment */
    /* ut : list of transactions accessing only second vertical fragment */
    /* it : list of transactions accessing both 1st and 2nd vertical fragments */
    lt = ut = it = NULL;
    for (k=0; k<ntr; k++)
        if (use(k,&h1)&&!use(k,&h2))
            append(&lt, k);
        else if (use(k,&h2)&&!use(k,&h1))
            append(&ut, k);
        else
            append(&it, k);
    /* compute the total access cost for all transactions */
    w1 = access(&lt) + access(&ut) + 2*access(&it);
    prez = MAXCOST;
    /* choose the best allocation sites bs and bt for 1st and 2nd vertical */
    /* fragments */
    for (s=0; s<nsite; s++)
        for (t=0; t<nsite; t++)
            if (s != t) {
                /* compute total transmission cost for all transactions if sites s and t */
                /* are chosen for 1st and 2nd vertical fragments */
                w3 = trans(&lt,&h1, s) + trans(&ut,&h2, t) + trans(&it,&h1, s) + trans(&it,&h2, t);
                z = c1*w1 + c3*w3;
                printf("        If %d and %d are allocation sites for vertical\n", s+1,t+1);
                printf("        fragments L and U, respectively, then the total\n");
                printf("        transaction execution cost is %d\n\n\n", z);
                if (z < prez) {
                    prez = z;
                    bs = s;
                    bt = t;
                }
            }
    printf("    * So :\n");
```

```
154         printf("Vertical fragments L and U should be allocated to\n");
155         printf("to sites %d and %d, respectively\n", bs+1, bt+1);
156 }
157 /*-----------------------------------------------------------------*/
158
159 /* function to append num to list *head */
160 void append (head, num)
161 struct el **head;
162 int num;
163 {
164         struct el *p;
165         p = (struct el *) malloc(sizeof(struct el));
166         p->no = num;
167         p->next = *head;
168         *head = p;
169 }
170
171 /*-----------------------------------------------------------------*/
172 /* function to determine if transaction tr uses any attributes in the list pt*/
173 int use(tr, pt)
174 int tr;
175 struct el **pt;
176 {
177         struct el *p;
178         int flg;
179         p = *pt;
180         flg = 0;
181         while (p != NULL && !flg) {
182                 if (au[tr][p->no] != 0)
183                         flg = 1;
184                 p = p->next;
185         }
186         return(flg);
187 }
188 /*-----------------------------------------------------------------*/
189 /* function to compute the total number of accesses made by all */
190 /* transactions in tlist */
191 int access(tlist)
192 struct el **tlist;
193 {
194         struct el *p;
195         int j, w;
196         p = *tlist;
197         w = 0;
198         while (p != NULL) {
199                 for (j=0; j<nsite; j++)
200                         w = w +f[p->no][j]*a[p->no];
201                 p = p->next;
202         }
203         return(w);
204 }
```

```
205
206 /*-------------------------------------------------------------------------*/
207 /* function to compute the total number of transmission bytes for all */
208 /* transactions in htr accessing vertical fragment hattr when that fragment */
209 /* is allocated at site */
210 int trans(htr, hattr, site)
211 struct el **htr, **hattr;
212 int site;
213 {
214         struct el *p1, *p2;
215         int j, attrlen, acc, w;
216         p1 = *htr;
217         p2 = *hattr;
218         w = 0;
219         while (p1 != NULL) {
220                 attrlen = 0;
221                 acc = 0;
222                 p2 = *hattr;
223                 while (p2 != NULL) {
224                         if (au[p1->no][p2->no] != 0)
225                                 attrlen = attrlen + leng[p2->no];
226                         p2 = p2->next;
227                 }
228                 for (j=0; j<nsite; j++)
229                         if (j != site)
230                                 acc = acc + f[p1->no][j]*a[p1->no];
231                 w = w + acc*attrlen;
232                 p1 = p1->next;
233         }
234         return(w);
235 }
```

PROGRAM TO FIND THE BEST ALLOCATION SITES FOR THE
TWO VERTICALLY PARTITIONED FRAGMENTS OF A RELATION

METHOD : NAVF Method 1 of Chapter 1, Major Report
(Non-replicated Allocation of Vertical Fragments)

SAMPLE INPUT :

. Attributes of relation R :
1 2 3 4 5 6 7 8 9 10

. Attributes of the first vertical fragment L :
1 5 6 7 8 9 10

. Attributes of the second vertical fragment U :
2 3 4

. Access cost per tuple, storage cost per byte, and
transmission cost per byte :
100 10 1000

. Lengths of attributes of R :
40 8 4 6 15 14 3 5 9 12

. Transaction access vector (each access retrieves
or updates one tuple) :
3 7 15 10 6 5 8 12

. Transaction frequency matrix :

10	15	0	0
10	20	10	10
0	0	15	10
10	15	0	10
5	10	5	5
10	5	5	5
5	10	5	5
5	5	2	3

. Attribute usage matrix :

1	0	0	0	1	0	1	0	0	0
0	1	1	0	0	0	0	1	1	0
0	0	0	1	0	1	0	0	0	1
0	1	0	0	0	0	1	1	0	0
1	1	1	0	1	0	1	1	1	0
1	0	0	0	1	0	0	0	0	0
0	0	1	0	0	0	0	0	1	0
0	1	1	1	0	0	0	0	0	0

OUTPUT :

If 1 and 2 are allocation sites for vertical fragments L and U, respectively, then the total transaction execution cost is 35698000

If 1 and 3 are allocation sites for vertical fragments L and U, respectively, then the total transaction execution cost is 37556000

If 1 and 4 are allocation sites for vertical fragments L and U, respectively, then the total transaction execution cost is 36990000

If 2 and 1 are allocation sites for vertical fragments L and U, respectively, then the total transaction execution cost is 34663000

If 2 and 3 are allocation sites for vertical fragments L and U, respectively, then the total transaction execution cost is 34761000

If 2 and 4 are allocation sites for vertical fragments L and U, respectively, then the total transaction execution cost is 34195000

If 3 and 1 are allocation sites for vertical fragments L and U, respectively, then the total transaction execution cost is 33873000

If 3 and 2 are allocation sites for vertical fragments L and U, respectively, then the total transaction execution cost is 32113000

If 3 and 4 are allocation sites for vertical fragments L and U, respectively, then the total transaction execution cost is 33405000

- If 4 and 1 are allocation sites for vertical fragments L and U, respectively, then the total transaction execution cost is 35023000

- If 4 and 2 are allocation sites for vertical fragments L and U, respectively, then the total transaction execution cost is 33263000 .

- If 4 and 3 are allocation sites for vertical fragments L and U, respectively, then the total transaction execution cost is 35121000

* So :
Vertical fragments L and U should be allocated to to sites 3 and 2, respectively

APPENDIX C

Program Listing for DATAID-1 Method for Secondary Index Selection

```
/*****************************************************************************/
/* PURPOSE : This program selects the best set of secondary indices (i.e. the*/
/*           one that results in the least total transaction execution cost) */
/*           built over the relations of a relational database               */
/* INPUT : . Transaction access matrix                                       */
/*         . Index update matrix                                             */
/*         . Index memory vector                                             */
/*         . Transaction primary access vector                               */
/*         . Transaction weight vector                                       */
/* OUTPUT : The best set of secondary indices                                */
/* METHOD : DATAID-1 Algorithm presented in chapter 3, Major Report          */
/*****************************************************************************/
#define MAXQ 100 /* max. no. of transactions */
#define MAXI 100 /* max. no of indices */
#define MAXR 50 /* max. no. of relations */
#define MD 1000 /* max. no. of data pages for indices */
#include <stdio.h>
FILE *fp;
struct bel {
        int indno;
        struct bel *next;
};
int hpos, vpos, next_hpos, next_vpos;
int p[MAXQ], mem[MAXI], w[MAXQ];
int ga[MAXQ][MAXI], gu[MAXQ][MAXI];
/*---------------------------------------------------------------------------*/
main()
{
        void fill(), order(), printindex();
        int minimum(), exist();
        FILE *fopen();
        int nrel, fdim, sdim, i, j, k, mb, tmb, pcost, tcost, primecost;
        int path[MAXQ], tpath[MAXQ], iset[MAXI], b[MAXQ], tb[MAXQ], dim[MAXR];
        struct bel *bset, *tbset, *ptr, *preptr;
        printf("          PROGRAM TO SELECT THE BEST SET OF SECONDARY INDICES\n");
        printf("              OVER THE RELATIONS OF A RELATIONAL DATABASE\n\n\n");
        printf("METHOD : DATAID-1 Algorithm, Chapter 3, Major Report\n\n\n");
        fp = fopen("dfile", "r");
        if (fp == NULL) {
                fprintf(stderr, "cannot read the file\n");
                exit(1);
        }
        fscanf(fp, "%d", &nrel);
        hpos = vpos = 0;
        printf("SAMPLE INPUT :\n\n");
        for (k = 0; k < nrel; k++) {
                printf("   * Relation %d\n", k+1);
                fscanf(fp, "%d%d", &fdim, &sdim);
                if (fdim==0)
                        dim[k] = sdim;
                else
```

```
52                      dim[k] = dim[k-1] + sdim;
53              next_hpos = hpos + fdim;
54              next_vpos = vpos + sdim;
55              /* fill in transaction primary access vector */
56              printf("        . Transaction primary access vector :\n");
57              printf("          ");
58              for (i = hpos; i < next_hpos; i++) {
59                      fscanf(fp, "%d", &p[i]);
60                      printf("%d    ", p[i]);
61              }
62              printf("\n\n");
63              /* fill in index memory vector */
64              printf("        . Memory vector for indices :\n");
65              printf("          ");
66              for (j = vpos; j < next_vpos; j++) {
67                      fscanf(fp, "%d", &mem[j]);
68                      printf("%d    ", mem[j]);
69              }
70              printf("\n\n");
71              printf("        . Transaction access cost vector :\n");
72              printf("          ");
73              fill(ga, 'a'); /* fill in global transaction access cost matrix */
74              printf("\n\n");
75              printf("        . Index update cost vector :\n");
76              printf("          ");
77              fill(gu, 'u'); /* fill in global index update cost matrix */
78              printf("\n\n");
79              /* fill in transaction weight vector */
80              printf("        . Transaction weight vector :\n");
81              printf("          ");
82              for (i = hpos; i < next_hpos; i++) {
83                      fscanf(fp, "%d", &w[i]);
84                      printf("%d    ", w[i]);
85              }
86              printf("\n\n");
87              hpos = next_hpos;
88              vpos = next_vpos;
89      }
90      order(iset); /* order indices according to increasing global transaction */
91      /* execution costs */
92      for (i = 0; i < hpos; i++)
93              path[i] = -1;
94      mb = 0;
95      bset = NULL;
96      pcost = 0;
97      for (i = 0; i < hpos; i++)
98              pcost = pcost + w[i]*p[i];
99      primecost = pcost;
100     printf("\n\n\nOUTPUT :\n\n");
101     printf("   * The cost without any indexes is : %d\n\n", primecost);
102     /* find the best set of secondary indices */
```

```
103             for (j = 0; j < vpos; j++) {
104                     ptr = (struct bel *) malloc(sizeof(struct bel));
105                     ptr->indno = iset[j];
106                     ptr->next = bset;
107                     tbset = ptr;
108                     tmb = mb + mem[iset[j]];
109                     for ( i = 0; i < hpos; i++)
110                             if (path[i] == -1)
111                                     if (ga[i][iset[j]] < p[i])
112                                             tpath[i] = iset[j];
113                                     else
114                                             tpath[i] = -1;
115                             else
116                                     if (ga[i][iset[j]] < ga[i][path[i]])
117                                             tpath[i] = iset[j];
118                                     else
119                                             tpath[i] = path[i];
120                     /* remove redundant indices in tbset */
121                     if (exist(tpath, iset[j])) {
122                             ptr = preptr = tbset;
123                             while (ptr != NULL)
124                                     if (ptr->indno != iset[j])
125                                             if (!exist(tpath, ptr->indno)) {
126                                                     tmb = tmb - mem[ptr->indno];
127                                                     preptr->next = ptr->next;
128                                                     free(ptr);
129                                                     ptr = preptr->next;
130                                             }
131                                             else {
132                                                     preptr = ptr;
133                                                     ptr = ptr->next;
134                                             }
135                                     else {
136                                             preptr = ptr;
137                                             ptr = ptr->next;
138                                     }
139                             for (i = 0; i < hpos; i++)
140                                     tb[i] = minimum(i, tbset);
141                             tcost = 0;
142                             for (i = 0; i < hpos; i++)
143                                     tcost = tcost + w[i]*tb[i];
144                             if (tcost < pcost && tmb < MD) {
145                                     bset = tbset;
146                                     mb = tmb;
147                                     for (i = 0; i < hpos; i++) {
148                                             path[i] = tpath[i];
149                                             b[i] = tb[i];
150                                     }
151                                     pcost = tcost;
152                             }
153             }
```

```
154         }
155         printf("   * If the following indexes are defined : \n");
156         printindex(bset, dim);
157         printf("     then the total cost is : %d\n\n", pcost);
158         printf("   * So the cost reduction is : %d percent\n",(primecost-pcost)*100/primecost);
159 }
160 /*-------------------------------------------------------------------------*/
161 /* this function fills in the global transaction access cost matrix or */
162 /* the global index update cost matrix */
163 void fill(table, type)
164 int table[][MAXI];
165 char type;
166 {
167         int i, j;
168         for (i = hpos; i < next_hpos; i++) {
169                 for (j = vpos; j < next_vpos; j++) {
170                         fscanf(fp, "%d", &table[i][j]);
171                         printf("%d   ", table[i][j]);
172                 }
173                 printf("\n          ");
174         }
175         for (i = 0; i < hpos; i++)
176                 for (j = vpos; j < next_vpos; j++)
177                         if (type =='a')
178                                 table[i][j] = p[i];
179                         else
180                                 table[i][j] = 0;
181         for (i = hpos; i < next_hpos;i++)
182                 for (j = 0; j < vpos; j++)
183                         if (type == 'a')
184                                 table[i][j] = p[i];
185                         else
186                                 table[i][j] = 0;
187 }
188 /*-------------------------------------------------------------------------*/
189 /* this function orders secondary indices according to increasing global */
190 /* transaction execution costs */
191 void order(vector)
192 int vector[];
193 {
194         int i, j, l, low;
195         struct el {
196                 int cost;
197                 int ind;
198         };
199         struct el temp[MAXI], t;
200         for (j = 0; j < vpos; j++) {
201                 temp[j].cost = 0;
202                 for (i = 0; i < hpos; i++)
203                         temp[j].cost = temp[j].cost + w[i]*(ga[i][j] + gu[i][j]);
204                 temp[j].ind = j;
```

```
205             }
206             for (j =0; j < vpos-1; j++) {
207                     low = j;
208                     for (l = j +1; l < vpos; l++)
209                             if (temp[low].cost > temp[l].cost)
210                                     low = l;
211                     if (low != j) {
212                             t.cost = temp[low].cost;
213                             t.ind = temp[low].ind;
214                             temp[low].cost = temp[j].cost;
215                             temp[low].ind = temp[j].ind;
216                             temp[j].cost = t.cost;
217                             temp[j].ind = t.ind;
218                     }
219             }
220             for (j = 0; j < vpos; j++)
221                     vector[j] = temp[j].ind;
222     }
223     /*-------------------------------------------------------------------------*/
224     /* this function finds the best execution cost to perform a transaction */
225     int minimum(q, bptr)
226     int q;
227     struct bel *bptr;
228     {
229             struct bel *pt;
230             int mini;
231             mini = p[q];
232             pt = bptr;
233             while (pt != NULL) {
234                     if (mini > ga[q][pt->indno])
235                             mini = ga[q][pt->indno];
236                     pt = pt->next;
237             }
238             pt = bptr;
239             while (pt != NULL) {
240                     mini = mini + gu[q][pt->indno];
241                     pt = pt->next;
242             }
243             return(mini);
244     }
245     /*-------------------------------------------------------------------------*/
246     /* this function tests if an index exists in a vector */
247     int exist(vect, q)
248     int vect[];
249     int q;
250     {
251             int i = 0;
252             int t = 0;
253             while (!t && i < hpos) {
254                     if (vect[i] == q)
255                             t = 1;
```

```
256                     i++;
257             }
258             return(t);
259     }
260     /*-------------------------------------------------------------------------*/
261     /* this function prints the best set of secondary indices */
262     void printindex(pter, v)
263     struct bel *pter;
264     int v[];
265     {
266             int i, test;
267             struct bel *pt;
268             pt = pter;
269             while (pt != NULL) {
270                     i = 0;
271                     test = 0;
272                     while (!test) {
273                             if (pt->indno < v[i])
274                                     test = 1;
275                             i++;
276                     }
277                     if (i ==1)
278                             printf("       . Index %d in relation %d\n", pt->indno + 1, i);
279                     else
280                             printf("       . Index %d in relation %d\n", (pt->indno - v[i-2])+1, i);
281                     pt = pt->next;
282             }
283     }
```

PROGRAM TO SELECT THE BEST SET OF SECONDARY INDICES
OVER THE RELATIONS OF A RELATIONAL DATABASE

METHOD : DATAID-1 Algorithm, Chapter 3, Major Report

SAMPLE INPUT :

* Relation 1
  . Transaction primary access vector :
    62 40 40

  . Memory vector for indices :
    10 5

  . Transaction access cost vector :
    13 62
    18 14
    40 40

  . Index update cost vector :
    0 10
    0 0
    10 10

  . Transaction weight vector :
    100 50 20

* Relation 2
  . Transaction primary access vector :
    67 25 40

  . Memory vector for indices :
    10 8 6

  . Transaction access cost vector :
    17 67 27
    11 6 21
    40 40 15

  . Index update cost vector :
    0 20 0
    0 0 0
    8 8 0

  . Transaction weight vector :

100 200 50

OUTPUT :

* The cost without any indexes is : 22700

* If the following indexes are defined :
  . Index 1 in relation 1
  . Index 3 in relation 2
  . Index 1 in relation 2
  then the total cost is : 8250

* So the cost reduction is : 63 percent