The Partial Normalized Storage Model of Nested Relations *

Aladdin Hafez# and Gultekin Ozsoyoglu

Department of Computer Engineering and Science
Case Western Reserve University
Cleveland, Ohio 44106

ABSTRACT

This paper introduces the partial normalized storage model of nested relations which uses the workload information of the database system under consideration to obtain a "better" storage model (i.e., one with a lower query cost) for a given nested relation. Based on the normalized storage model, the nested relation scheme is graphically represented as a tree called the scheme tree. By using the workload information, and by performing a series of merges on the nodes of the scheme tree, a near-optimum scheme tree is produced to represent the partial normalized storage model.

We prove that our approach which uses the greedy method, locates the optimum scheme tree in most of the cases. In few cases, when the approach locates a "near" optimum scheme tree, the relative difference between the costs of the produced scheme tree and the optimum scheme tree is shown to be very small.

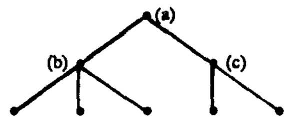
1. Introduction

Supporting new database applications in fields such as computer aided design, computer aided manufacturing and artificial intelligence requires efficient implementations of *nested relations* (i.e., relations containing relations, also called *non-first-normal-form (N1NF) relations* or *complex objects*). Storage structures for nested relations have been investigated by many researchers [HamN79, AbiB84, DKAB86, StoR86, VaKC86, KiCB87, DesV88].

Example 1: As a running example in this paper, we use the nested relation scheme R where R=(a, A, B), A=(b, C, D, E), B=(c, F, G), C=(d, e), D=(f, g), E=(h, i), F=(j, k) and G=(l, m). R, the outermost relation, is called the *external relation.* A, B, C, D, E, F and G are *internal relations* (i.e., relations inside the external relation). *Atomic attributes* (i.e., attributes whose values are single values) are denoted by lower-case letters. The scheme of R and a single tuple in R are given in figure 1.

	a	A								B				
Attributes		b	C		D		E		c	F		G		
			d	e	f	g	h	i		j	k	l	m	
A single external relation tuple	a_1	b_1	d_1	e_1	f_1	g_1	h_1	i_1	c_1	j_1	k_1	l_1	m_1	
			d_2	e_2	f_2	g_2	h_2	i_2		j_2	k_2	l_2	m_2	
		b_2	d_3	e_3	f_3	g_3	h_3	i_3	c_2	j_3	k_3	l_3	m_3	
			d_4	e_4	f_4	g_4	h_4	i_4		j_4	k_4	l_4	m_4	

A single tuple in R
(a)



Scheme tree of R
(b)
Figure 1

In the literature, System 2000 and ADABAS [Olle71] are two hierarchical database systems used to implement nested relations. The segments of hierarchical records are inserted into one file. All hierarchical relationships are expressed by a second file. In OASIS [Wied83], an instance of a tuple, together with its descendants, is placed into a single compact variable-length record.

Dadam and others use the technique of separating structural information from data [DKAB86], where each external relation tuple is stored into one record, and, for each tuple, there is an entry in a directory, called the *mini-directory* (MD). The mini-directory is used to handle the allocation of each tuple.

In IMS [McGe77], a relation and all its internal relations appear in a single file. Each top-level entry in a file contains the atomic attributes of a tuple in the external relation. The four file options for IMS are

HSAM, HISAM, HDAM and HIDAM.

Recently, some new database management systems supporting nested relations have been designed for non-traditional applications areas. EXODUS [CDRS86] and POSTGRES [StoR86] are two examples. In EXODUS, the basic unit of stored data is the storage object (an entire tuple). Storage objects can grow and shrink in size without putting any restrictions on where we should delete or insert. Accordingly, the system supports insertion and deletion of new portions of a storage object anywhere within the object. In POSTGRES, a new datatype, called POSTQUEL, has been defined to support nested relations. A field of type POSTQUEL contains a sequence of commands to retrieve data from other relations that represent the subobjects. All relations are stored as heaps within an optional collection of secondary indices.

In the literature, storage models of nested relations are in general classified into four storage models [HofS75, NCWJ84, CopK85, DKAB86, StoR86, VaKC86, KCJB87]. Below we describe how each storage model represents a nested relation. To demonstrate each storage model, we use the nested relation scheme R of example 1. Assume that, for each external/internal relation Φ there is a *surrogate attribute* S_Φ, (i.e., an artificial identifier) which is used as an identifier for relation Φ (similar to "tuple identifiers" in 1NF relations).

The *Decomposed Storage Model* (DSM) [CopK85, KCJB87] utilizes a transposed storage. Each atomic attribute of a relation with a surrogate for record identity forms a binary relation. Each binary relation is stored into a separate file.

Example 2: (DSM) There are 13 binary relations and thus 13 files in the DSM representation of R. The binary relations are:

$R_a=(S_R,a)$, $A_b=(S_A,b)$, $B_c=(S_B,c)$, $C_d=(S_C,d)$, $C_e=(S_C,e)$, $D_f=(S_D,f)$, $D_g=(S_D,g)$, $E_h=(S_E,h)$, $E_i=(S_E,i)$, $F_j=(S_F,j)$, $F_k=(S_F,k)$, $G_l=(S_G,l)$ and $G_m=(S_G,m)$

The *Normalized Storage Model* (NSM) [StoR86, VaKC86] decomposes a nested relation in such a way that the atomic attributes of each external/internal relation tuple form a record of a file, and inner relations at a given level are related to each other by using surrogates. Join indices [ValB86, Vald87] may be used in order to retrieve an internal relation.

Example 3: (NSM) There are 8 files in the NSM representation of R. Each file contains the atomic attributes of an internal/external relation along with its surrogate attribute. The records of each file have the following format.

$R=(S_R,a)$, $A=(S_A,b)$, $B=(S_B,c)$,$C=(S_C,d,e)$, $D=(S_D,f,g)$, $E=(S_E,h,i)$, $F=(S_F,j,k)$ and $G=(S_G,l,m)$

The *Flattened Storage Model* (FSM) [DKAB86, VaKC86] is originally called the direct storage model. A nested relation is stored directly into a file. Each record of a file represents an entire external relation tuple. The record of a file can be clustered on atomic attributes of the external relation tuples. Access to nested relation tuples based on attributes other than those of the external relation tuples are done by using secondary indices or sequential scans.

Example 4: (FSM) The whole nested relation instance is stored into one file whose record format is

$(S_R,a,\{S_A,b,\{S_C,d,e\},\{S_D,f,g\},\{S_E,h,i\}\}, \{S_B,c,\{S_F,j,k\},\{S_G,l,m\}\})$

The DSM and the NSM may be viewed as special cases of another storage model. The *Partial Decomposed Storage Model* (P-DSM) [HofS75, NCWJ84, VaCK86] is a mix between the DSM and the NSM. The atomic attributes of an internal/external relation are partitioned such that those atomic attributes that are frequently accessed together are stored in the same file. Each file contains a set of atomic attributes and a surrogate of their conceptual relation.

Example 5: (P-DSM) The P-DSM model for R can take several possible forms. One of those forms produces 10 files with the record formats

$R=(S_a,a)$, $A=(S_A,b)$, $B=(S_B,c)$, $C=(S_C,d,e)$, $D_1=(S_D,f)$, $D_2=(S_D,g)$, $E=(S_E,h,i)$, $F_1=(S_F,j)$, $F_2=(S_F,k)$ and $G=(S_G,l,m)$

We now introduce a new type of storage model, the *Partial Normalized Storage Model* (P-NSM). The NSM and the FSM may be viewed as two special cases of the P-NSM. In the P-NSM, the nested relation is vertically partitioned such that those subobjects (i.e., internal relations) which are frequently accessed together are stored in the same file. Each file contains the atomic attributes of an internal/external relation and some of its descendants. Figure 2 and example 6 illustrate the P-NSM.

Example 6: (P-NSM) One possible P-NSM model for R may have 6 files with formats

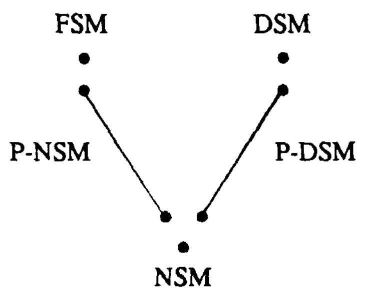


Figure 2. The Spectrum of Different Storage Models for Nested Relations.
R=(S_R,a,{S_A,b}),B=(S_B,c,{S_F,j,k}), C=(S_C,d,e), D=(S_D,f,g), E=(S_E,h,i) and G=(S_G,l,m)

By storing into the same file the internal relations which are accessed together frequently, the number of I/O operations required to respond to queries can be reduced. In general, the storage model of a nested relation can be classified as supporting three different query types: (a) queries manipulating entire external relation tuples, (b) queries manipulating internal relations and (c) queries manipulating specific individual components of (external/internal) relation tuples and their atomic attributes at different nesting levels. If the query type (a) is the dominant query type then clearly the FSM is the best choice to implement the nested relation. For the query type (b), the NSM is the obvious choice to implement the nested relation. Query type (c) constitutes the general case for nested relation queries. The P-NSM provides good support for those nested database systems which have the query type (c) as the dominant query type. By using the appropriate methodology and the workload information of the system, the NSM may be turned into a P-NSM with a better query processing performance. This paper describes such a methodology.

The NSM is graphically represented as a tree called the *scheme tree*. Each node in the scheme tree represents a file containing the atomic attributes of an internal/external relation. Figure 1-(b) describes the scheme tree of R. In this paper, we are specifically concerned with two parameters of the workload information of a database system. *Query count* is the first parameter. Each node and each edge in the scheme tree has its own query count (frequency) denoting the number of queries that manipulate the information in that node and in the nodes adjacent to that edge, respectively. In this paper, we introduce the ASSIGN-F algorithm which, given a set of queries, assigns frequencies to the nodes and edges in the scheme tree in a consistent manner. The second parameter is the *query cost*. The query cost of each node is a function of the size of the file represented by the node, and the type of the query used. Each node may have a number of different query costs. Queries are classified according to their disk processing costs. Each group of queries contains those queries that have the same disk processing cost.

We compute the *scheme tree cost* by using the query costs and the frequencies of the nodes and the edges of the scheme tree. Depending on the query types, the cost of the scheme tree in the NSM can be changed by reducing the depth of the scheme tree through the *merge operation* of two or more nodes (which produces a P-NSM representation). Then, the scheme tree with the lowest cost can be found by enumerating all the possible trees derivable from the scheme tree by the merge operation.

In this paper, we introduce the GREEDY-MERGE algorithm which takes the original NSM scheme tree, the query types, the associated query costs for each node, and frequencies for each node and for each edge in the scheme tree, and uses the greedy method to convert the NSM scheme tree into a P-NSM scheme tree with a lower scheme tree cost. The GREEDY-MERGE algorithm utilizes a level order traversal starting at the lowest level. At each step, a subset of nodes (at the present level) are checked for merging them into their father node. After examining the nodes at a certain level, the algorithm moves up to the next higher level and repeats the merge checks at the new level. For a large number query cost types, the GREEDY-MERGE algorithm finds the scheme tree with the lowest cost, without having to enumerate all the possible trees derivable from the scheme tree.

The preliminary experiments in section 4.2 show that out of 25,000 cases, the GREEDY-MERGE algorithm have produced P-NSM structures with the optimal scheme trees in 95.2% of the cases. For those cases where a suboptimal scheme tree were obtained, the average and the maximum percentage of error were 1.255% and 5.744%, respectively.

The rest of the paper is organized as follows. Section 2 presents the scheme tree model and the ASSIGN-F algorithm which is used to assign frequencies to the nodes and edges in the scheme tree. In section 3, we analyze some special cases of scheme trees for their optimum scheme tree costs, and then introduce the GREEDY-MERGE algorithm. Section 4 discusses the preliminary experimental results.

2. Assigning Frequencies to The Scheme Tree

In this section, we present the scheme tree model and its frequency assignment algorithm. Section 2.1 introduces the scheme tree model. The

ASSIGN-F algorithm is given in section 2.2. Section 2.3 describes the MERGE procedure along with the frequency adjustments of nodes and edges due to the merges in the scheme tree.

2.1. Scheme Tree Model

The *atomic relation* R_A of a (internal or external) relation R is the relation which contains only the atomic attributes of R. Clearly, for a *first-normal-form* (1NF) *relation* S, the corresponding atomic relation is S itself. To illustrate, consider a nested relation scheme R, where R=(a, b, B, C), B=(a_1, b_1) and C=(a_2, b_2). Then, R_A=(a, b), B_A=B and C_A=C are the atomic relations of relations R, B and C, respectively.

The scheme tree T with the node set N_T of a nested relation represents the nesting structure of the relation. The scheme tree is also the graphical representation of the NSM. Each node n_i in N_T represents a file which contains the atomic relation of that (internal or external) relation. Each edge in the scheme tree T represents a relationship between two atomic relations of the nested relation that are adjacent to the edge.

Example 7: Consider the nested relation scheme R of example 1. The NSM of R consists of the eight atomic relations R_A=(a), A_A=(b), B_A=(c), C_A=(d, e), D_A=(f, g), E_A=(h, i), F_A=(j, k) and G_A=(l, m). The scheme tree T of the nested relation R is given in figure 1 (b). For notational simplicity, we will use nodes n_R, n_A, n_B, n_C, n_D, n_E, n_F and n_G to represent (the scheme tree nodes and) the files which contain the atomic relations R_A, A_A, B_A, C_A, D_A, E_A, F_A and G_A,respectively.

We now assign two parameters to the nodes and edges in a given scheme tree:

a) the query cost of a node (file), and
b) the query frequency of a node and an edge.

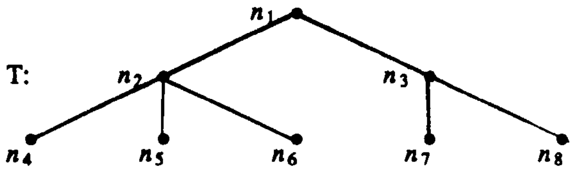
The query cost of a node depends on the types of queries accessing that node, the file organization of the node and the size of the file represented by that node. Query costs will be discussed in section 3.

Query frequencies of nodes and edges depend on the relationship between the nodes involved in the queries. We first give some definitions and examples.

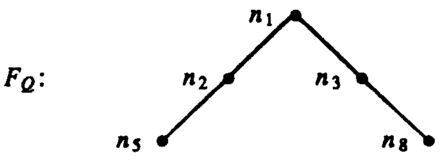
Definition (*Query Node, Nonquery Node*). For a query Q, a node n is called a *query node* if it is referred to in that query. N_Q is the set of all query nodes with respect to the query Q. Those nodes which do not appear in the query Q are called *nonquery nodes*. N_T-N_Q is the set of nonquery nodes with respect to query Q.

Definition (*Query Forest, Query Tree*). Delete the nonquery nodes from a scheme tree T. The remaining set of trees form a *query forest* F_Q with respect to query Q. If the cardinality of F_Q is one (i.e., $|F_Q|$=1) then the resulting tree is called the *query tree* T_Q.

Example 8: Let the scheme tree T with the node set N_T be

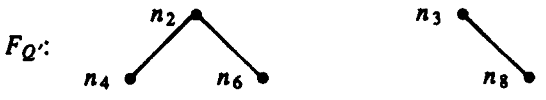


Assume for a given query Q, the set of query nodes is N_Q={n_1, n_2, n_3, n_5, n_8}. Then the set of nonquery nodes is {n_4, n_6, n_7}. If the nonquery nodes are removed from the scheme tree T then the remaining nodes will form the following query forest :



which contains a single tree T_Q.

Assume for a given query Q', the set of query nodes is $N_{Q'}$= {n_2, n_3, n_4, n_6, n_8}. Then the set of nonquery nodes is {n_1, n_5, n_7}. If the nonquery nodes are removed from the scheme tree then the remaining nodes will form the following query forest $F_{Q'}$



$F_{Q'}$ consists of two distinct trees, i.e., $|F_{Q'}|$=2.

Assumption: In the rest of the paper, we consider only those queries with $|F_Q|$=1. Thus, T_Q of a given query Q is unique.

We think that most (if not all) "meaningful" nested relation queries satisfy the above assumption. We give an example.

Example 9: Let the nested relation scheme DEPT-EMP be

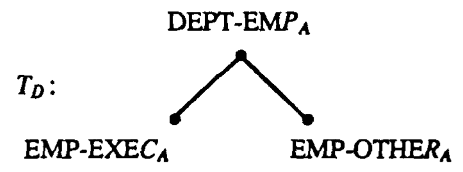
DEPT-EMP = (dno, dname, EMP-EXEC, EMP-OTHER),

where dno and dname are atomic attributes. EMP-EXEC and EMP-OTHER are higher order attributes. EMP-EXEC is the relation of executive employees while EMP-OTHER is the relation of other employees.

EMP-EXEC = (eno, ename, salary, success-ratio, . . .) and

EMP-OTHER = (eno, ename, salary, children, . . .).

The scheme tree T_D of the nested relation DEPT-EMP is

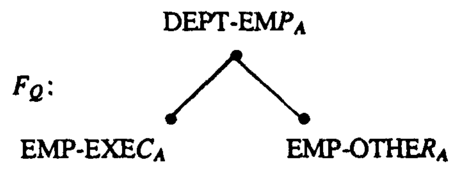


Let a query Q be " Select those employees which have salary <20 K ".

Query Q could be written as

$\sigma_{salary<20K}$ ($\pi_{EMP-EXEC}$(DEPT-EMP) $\cup$ $\pi_{EMP-OTHER}$ (DEPT-EMP)),

which has the query forest F_Q



where $|F_Q|$=1.

A file F represented by the scheme tree node n may be accessed by

a) queries that only refer to attributes in n, and/or

b) queries that refer to n and other nodes n_i which are adjacent to n.

Our goal is to take a scheme tree and modify it into a new "scheme tree" for a better performance. To this end, rather than directly counting query counts of files, we assign frequencies to both nodes and edges of the scheme tree in order to compute access counts of files for the revised scheme trees. The *frequency f_j of node n_j* for query type q denotes the count of those queries of type q that only refer to the attributes in n_j. The *frequency fx_i of edge e_i* for query type q incident to nodes n and n_j denotes the count of those queries of type q that refer to attributes of n *and* n_j. Assume there are L edges incident to node n in T and e_i, $1 \le i \le L$, is an edge incident to node n. The *query count $A_{n,q}$ of the file* F with respect to query type q denotes the count of queries of type q that refer to attributes of n. Thus $A_{n,q}$ is defined as

$$A_{n,q} := (\sum_{i=1}^{L} fx_i) + f_n$$

2.2. Frequency Assignment Algorithm

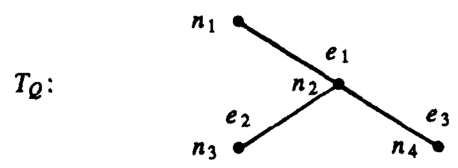
The algorithm ASSIGN-F given in figure 3 assigns frequencies to nodes and edges in T.

```
Algorithm ASSIGN-F (T,S)

 Input: (a)  A scheme tree T with some node and edge
             frequencies (possibly all frequencies are zero), and
        (b)  a sequence S of queries such that, for each query Q
             in S, $T_Q$ is unique,i.e.,$|F_Q|$=1.
 Output:     New frequency assignments to edges and nodes in T.
   begin
     {$fx_i$ denotes the frequency of edge $e_i$; $f_j$ denotes the
     frequency of node $n_j$}
     for each query Q in S do
       begin
         for each edge $e_i$ in $T_Q$ do $fx_i := fx_i + 1$;
         for each node $n_j$ in $T_Q$ do $f_j := f_j + 1 - d(n_j)$;
       end
   end.
```

Figure 3. Frequency assignment algorithm for a scheme tree

Example 10: Let T_Q of a query Q be



Before the incorporation of Q, assume f_1=3, f_2=2, f_3=5, f_4=2 and fx_1=1, fx_2=2, fx_3=4. According to the ASSIGN-F algorithm, the frequency of each edge e_i will be increased by 1. Thus, after the change, fx_1=2, fx_2=3 and fx_3=5. The frequency of each node n_j will be increased by 1- d (n_j) where d(n_j) denotes the degree of node n_j. Thus, after the change, f_1=3, f_2=0, f_3=5 and f_4=2.

Whenever a query Q refers to some attributes in a file, we would like to increment the query count of the file by 1. Lemma 1 below shows that the algorithm ASSIGN-F correctly performs this increment operation.

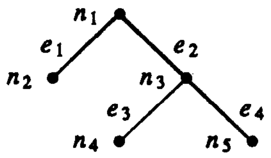
Lemma 1: Let $A_{n,q}$ be the query count of the file F represented by n. Assume there is a new query Q with query type q; $|F_Q| = 1$; n is a node in F_Q. After executing the algorithm ASSIGN-F, we have $A_{n,q} := A_{n,q}+1$.

Proof: Let the node n in the scheme tree T be also in F_Q, and d(n) be the degree of node n with respect to F_Q. Using the algorithm ASSIGN-F, the frequency of n will be increased by 1 - d (n), and the frequency of each edge adjacent to n in F_Q will be increased by 1. Since the number of edges which are adjacent to node n is d(n), the total sum of the frequency changes of node n and its adjacent edges equals (1 - d (n)) + d (n) = 1. Q.E.D.

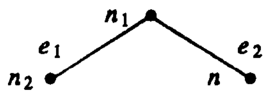
2.3. Adjusting Frequencies Under Changes to The Scheme Tree

Let T be a scheme tree and T_1 be a subtree of T. The algorithm MERGE$_{T_1}$(T) produces a new scheme tree T' from T by merging all the nodes of T_1 in T into a single node n. Thus the file represented by the new node n contains all the information in those files represented by the nodes in T_1. Then the frequency of node n in T', denoted by $f_n(T')$, is computed as the sum of the frequencies of all nodes n_j in T_1 and all edges e_j in T_1.

Example 11: Let the scheme tree T be

T:


Let T_1 denote the subtree with nodes n_3, n_4, n_5 and with edges e_3, e_4. If T_1 is reduced to a single node n by the algorithm MERGE$_{T_1}$(T) then the new scheme tree T' will be

T':


where

$f_j(T') = f_j(T),\ 1 \le j \le 2,$

$fx_i(T') = fx_i(T),\ 1 \le i \le 2$ and

$f_n(T') = \sum_{j=3}^{5} f_j(T) + \sum_{i=3}^{4} fx_i(T).$

Lemma 2 allows us to avoid recomputing frequency assignments from the original query sequence after each change in the scheme tree.

Lemma 2: For a scheme tree T, a subtree T_1 of T and a set S of queries,

MERGE$_{T_1}$(ASSIGN-F(T,S))=ASSIGN-F(MERGE$_{T_1}$(T),S)

Proof : By using enumeration and the fact that ASSIGN-F (ASSIGN-F (T, S_1), S_2) = ASSIGN-F (T, $S_1 \cup S_2$) for two disjoint sets S_1 and S_2 of queries.

3. Optimum Scheme Trees

This section discusses a technique to find a better (i.e., optimum or near-optimum) scheme tree. A new structure is established by merging two or more nodes in the original NSM scheme tree. The evaluation criteria of a given scheme tree T is the number disk accesses (characterized by the scheme tree cost) required to process the queries (i.e., the workload) using the files represented by the nodes in the scheme tree T. Thus, the *optimum scheme tree* for a given workload is the one with the lowest scheme tree cost. We present the GREEDY-MERGE algorithm which takes the original scheme tree, the query types, the associated query costs for each node, and the frequencies for each node and each edge in the scheme tree, and uses the greedy method to convert the scheme tree into a "near-optimum" scheme tree. To judge the performance of the GREEDY-MERGE algorithm, we also discuss a number of special cases of query costs for which we analytically find the optimum scheme tree.

3.1. The GREEDY-MERGE Technique

Query cost $C_{n,q}$ of node n *with respect to query type* q is the number of disk accesses required to process a single query with type q on the file F represented by node n. Each node may have a number of different query costs according to the types of queries involved in the database system under consideration. Queries are classified according to their disk processing costs. Each group or type of queries contains those queries which have the same disk processing cost. The disk processing cost is represented in the order notation as a function of the size of the associated file. Query costs for different query types may be $K_1.1$, $K_2.\log N$, $K_3.N$, . . ., etc., where N is the size of the file used to process the query and K_i's are constants.

We now generalize the model given in section 2 so that for each query type q, the system keeps different (node and edge) frequencies. Consider the scheme tree T with the set of nodes N_T, the query type q and the file F represented by node n in T. *The cost of file* F *with respect to the query type* q is defined as the query count of F for query type q times the query cost of F with respect to query type q. For a set of queries belonging to different query types QT, *the cost of* F *with respect to* QT is defined as

$$C_{n,QT} = \sum_{q \in QT} A_{n,q} * C_{n,q}$$

where $A_{n,q}$ and $C_{n,q}$ denote the query count and the query cost of F with respect to q, respectively.

Consider a scheme tree T with a set of nodes N_T and a set of query types QT. The *scheme tree cost* E_T with respect to QT is defined as

$$E_T = \sum_{n \in N_T} C_{n,QT}$$

Depending on the query types, the scheme tree cost changes when the nesting depth of the scheme tree changes. The depth of the scheme tree is reduced by merging two or more nodes together.

One way to find the optimum scheme tree (i.e., the one with the minimum scheme tree cost) is to enumerate all possible trees derivable from the scheme tree by merging nodes. For a scheme tree T with a node set N_T, there are $2^{|N_T|-1}$ possible trees derivable from the scheme tree T, where $|N_T|$ is the cardinality of N_T. Cleary such an approach is not feasible even for reasonably small $|N_T|$ values.

Another approach for finding a "near-optimum" scheme tree is as follows: A scheme tree T is a collection of two-level subtrees. Each two-level subtree consists of a "father" node and "child" nodes. For each two-level subtree, one can enumerate all possible merges to find the associated optimum subtree. The resulting overall scheme tree is clearly a "near-optimum" scheme tree. For each two-level subtree ST_i with the node set SN_i, $1 \le i \le NS$, where NS is the number of the two-level subtrees in the scheme tree T, there are $2^{|SN_i|-1}$ possible subtrees derivable from the subtree ST_i, where $|SN_i|$ is the cardinality of SN_i. In the whole scheme tree T, the number of subtrees examined by this approach is

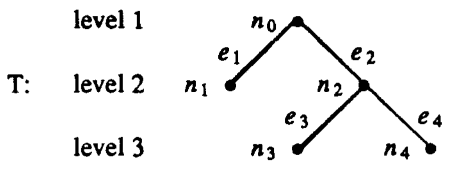
$$\sum_{i=1}^{NS} 2^{|SN_i|-1}$$

which may still be quite costly.

Our approach for finding the near-optimum scheme tree is to find a near-optimum subtree for each two-level subtree in the scheme tree T. For a given two-level subtree, the approach starts by calculating the costs of the subtrees resulted from merging one of the "child" nodes into the "father" node. Clearly the number of such subtrees equals the number of "child" nodes. We then compare the costs of the original subtree and the derived subtrees, and choose the one with the lowest cost to be the new subtree. Then the new subtree is once again investigated for a possible single-node-merge which produces a lower cost subtree and so on until there is no more cost improvement in the two-level scheme tree. In our approach described in figure 3 (the GREEDY-MERGE algorithm), the number of the examined subtrees is $O(|N_T|^2)$.

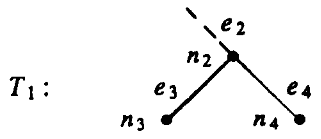
The GREEDY-MERGE algorithm utilizes a level order, left-to-right traversal starting at the lowest level. At each step, according to the cost formula used, each node at that level is checked for merging it into its father node. After examining the two-level subtrees at a certain level, we move up to the next higher level and repeat the merge checks at the new level. The following example shows how the GREEDY-MERGE algorithm works.

Example 12: Assume the scheme tree T is as follows



For simplicity, we assume a single query type is involved in this example, and thus, we will not mention query types in the cost formulas of this example. Each node n_i, $0 \le i \le 4$, has frequency f_i and query cost C_i, and each edge e_j, $1 \le j \le 4$, has frequency fx_j.

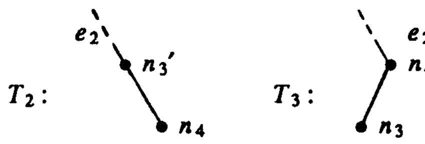
GREEDY-MERGE algorithm starts with the nodes at level 3. Subtree T_1 contains those nodes at level 3 along with their father node.



The scheme tree cost of the subtree T_1 is

$$E_{T_1} = C_2 (f_2 + \sum_{i=2}^{4} fx_i) + \sum_{i=3}^{4} C_i (f_i + fx_i)$$

The other alternatives are merging n_3 with n_2 (subtree T_2), and merging n_4 with n_2 (subtree T_3). For example, the new node n_3' in T_2 has the frequency $f_3' = f_2 + f_3 + fx_3$ and the query cost C_3' which is the query cost of the file produced by merging the files represented by nodes n_2 and n_3. The new subtrees T_2 and T_3 are :



The scheme tree cost of the subtrees T_2 and T_3 are

$$E_{T_2} = C_3' (f_2 + f_3 + \sum_{i=2}^{4} fx_i) + C_4 (f_4 + fx_4),$$

$$E_{T_3} = C_4' (f_2 + f_4 + \sum_{i=2}^{4} fx_i) + C_3 (f_3 + fx_3)$$

where C_4' is the query cost of the new node n_4', and the frequency of node n_4' is $f_4' = f_2 + f_4 + fx_4$.

According to the three cost formulas E_{T_1}, E_{T_2} and E_{T_3}, the GREEDY-MERGE algorithm chooses the subtree which has the lowest cost. In the case when E_{T_1} is the lowest cost, the GREEDY-MERGE algorithm moves up to the higher level and repeats the procedure. On the other hand, when either E_{T_2} or E_{T_3} is the lowest cost, the GREEDY-MERGE algorithm takes the subtree resulted from the last step (i.e, T_2 or T_3) and repeats the procedure. Figure 3 gives the GREEDY-MERGE algorithm.

The lemma below gives an upper bound to the number of subtrees examined by the GREEDY-MERGE algorithm.

Lemma 3: For a scheme tree T with a node set N_T, the GREEDY-MERGE algorithm evaluates, in the worst case, less than $|N_T| (|N_T| - 1) / 2$ different two-level subtrees of the scheme tree T.

Consider figure 4. Assume the subtree ST is a two-level subtee in a scheme tree T. Each node n_i, $0 \le i \le r$, has frequency f_i and cost C_i, and each edge e_j, $0 \le j \le r$, has frequency fx_j. We use the notation $C_{\{i_0, i_1, \ldots, i_m\}}$ for the query cost of the node resulted after merging the nodes $n_{i_0}, n_{i_1}, \ldots, n_{i_m}$. Similarly $C_{\{i\}}$ denotes the query cost of the node n_i. We also use the notation $E_{\{i_0, i_1, \ldots, i_m\}}$ to denote the scheme tree cost of the two-level subtree obtained after merging nodes $n_{i_0}, n_{i_1}, \ldots, n_{i_m}$. The notation $E_{\{i_0\}}$ denotes the scheme tree cost of the original two-level subtree with no mergings.

```
Algorithm GREEDY-MERGE ($T$,$T'$)
  Input :  A scheme tree T with frequencies and costs on the
           nodes and edges in T.
  Output:  A "near-optimum" scheme tree $T'$

  begin
    for i from the highest level of T to 1 do
    begin
      for each two level subtree ST with leaves at level i do
      begin
        repeat
          calculate the cost of the subtree ST;
          for each leaf j, $1 \le j \le k$, in the scheme tree ST do
          begin
            Obtain subtree $ST_j$ by merging j to its father
            node;
            Calculate the cost of $ST_j$;
          end;
          Let $ST_t$ denote the lowest cost $ST_j$, $1 \le j \le k$;
          if cost ($ST_t$) < cost ($ST$)
            then replace ST with $ST_t$ and mark
            "improvement"
            else mark "no improvement";
        until there is "no improvement";
      end;
    end;
  end.
```

Figure 3. The GREEDY-MERGE algorithm

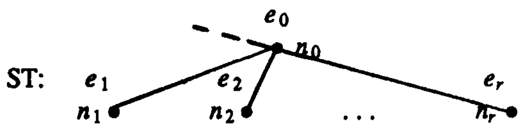


Figure 4.

The scheme tree cost of the subtree ST is

$$E_{\{0\}} = C_{\{0\}} (f_0 + \sum_{i=0}^{r} fx_i) + \sum_{i=1}^{r} C_{\{i\}} (f_i + fx_i)$$

Assume node n_j, $1 \le j \le r$, is merged with node n_0. Then the cost of the resulting subtree is

$$E_{\{0,j\}} = C_{\{0,j\}} (f_0 + f_j + \sum_{i=0}^{r} fx_i) + \sum_{i=1, i \ne j}^{r} C_{\{i\}} (f_i + fx_i)$$

Clearly, after the merge of n_j to n_0, the query costs of the immediate descendants of n_j will be effected by the merge. This is the reason that the GREEDY-MERGE algorithm produces a suboptimal scheme tree. In section 4, we discuss the effectiveness of the GREEDY-MERGE algorithm by comparing its performance with the optimal scheme trees for a number of selected types of scheme trees.

Let $K_l=\{0,i_1,i_2,\ldots,i_l\}$, $1 \le l \le r$ and $1 \le i_j \le r$. The cost E_{K_l} of the subtree ST after all the nodes n_i in K_l are merged to n_0 is

$$E_{K_l} = C_{K_l}\,(\sum_{i \in K_l} f_i + \sum_{i=0}^{r} fx_i) + \sum_{i \notin K_l} C_{\{i\}}\,(f_i + fx_i)$$

In the following section, we discuss some special cases of query costs for which we analytically find the optimum scheme tree.

3.2. Optimum Scheme Trees for Single Query Types

The evaluation of a nested relation manipulation operator (such as select, project, nest, etc.) depends on the file organizations of scheme tree nodes (e.g., sequential files, indexed files, hash files, . . . etc.) and the standard operations involved (e.g., search, sort, scan, . . . etc.). As an example, to evaluate a selection query, we need to search for a certain value(s). The number of disk accesses required to evaluate the search operations depends on the file organization. The processing cost may be O(1) (e.g., hash files), O(log N) (e.g., sequential sorted files), O(N) (e.g., heap files),

In this section, we assume that all the nodes have the same query type (and hence the same query cost). We discuss three different generic query costs for which the optimum scheme tree is either the original scheme tree (i.e., the NSM model) or a single node obtained by merging all the nodes in the scheme tree (i.e., the FSM model). The proofs of the lemmas in this section and section 3.3 are long [HafO88] and omitted due to space considerations.

Lemma 4 below shows that for those query types that have a constant query cost, the optimum scheme tree is the one that has all the nodes merged into a single node. That is the FSM is the best storage model for this case.

Lemma 4: Let $C_{n,q}$=a where a is a positive constant, $n \in N_T$, T is a scheme tree. Then the optimum scheme tree of T is a single node obtained by merging all the nodes in T. Moreover, the GREEDY-MERGE algorithm finds the optimum scheme tree.

Lemma 5 shows that when the common query cost of a node in the scheme tree is N times a monotonically increasing function of the associated file size (e.g., N log N or aN^k, k > 1) then the optimum scheme tree is the original scheme tree. That is, in such a case, the NSM is the best storage model for the scheme tree.

Lemma 5: Let $C_{n,q}$ = N g(N), $n \in N_T$, T is a scheme tree, N is the size of the file represented by node n, g(N) is a monotonically increasing function of N. Then the optimum scheme tree is the original scheme tree T. Moreover, the GREEDY-MERGE algorithm finds the optimum scheme tree.

3.3. A Sufficient Condition to Obtain the Optimum Scheme Subtree for Logarithmic Query Costs

In section 3.2, lemmas 4 and 5 have shown that when all nodes have the same query cost function, the GREEDY-MERGE algorithm always finds the optimum scheme tree for all practical query cost functions except log N, where N is the file size. This section shows that when the query cost $C_{n,q}$ is log N, the GREEDY-MERGE algorithm, under certain conditions, finds the optimum subtree for each two-level subtree in the scheme tree.

Let, after applying the GREEDY-MERGE algorithm, the subtree ST in figure 4 be transformed into the subtree ST_1 in figure 5, where m ≤ r.

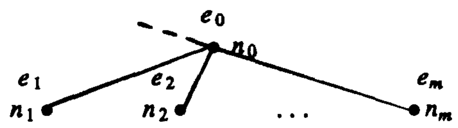


Figure 5

Then for any two nodes n_j and n_k, $1 \le j, k \le m$, the following two inequalities are always true,

$$C_{\{0\}}\,(f_0 + \sum_{i=0}^{m} fx_i) + C_{\{j\}}\,(f_j + fx_j) \le C_{\{0,j\}}\,(f_0 + f_j + \sum_{i=0}^{m} fx_i) \qquad (1)$$

$$C_{\{0\}}\,(f_0 + \sum_{i=0}^{m} fx_i) + C_{\{k\}}\,(f_k + fx_k) \le C_{\{0,k\}}\,(f_0 + f_k + \sum_{i=0}^{m} fx_i) \qquad (2)$$

However, it is not necessarily true that we have the inequality $E_{\{0\}} \le E_{\{0,j,k\}}$, which can be written as

$$C_{\{0\}}\,(f_0 + \sum_{i=0}^{m} fx_i) + C_{\{j\}}\,(f_j + fx_j) + C_{\{k\}}\,(f_k + fx_k) \le C_{\{0,j,k\}}\,(f_0 + f_j + f_k + \sum_{i=0}^{m} fx_i) \qquad (3)$$

We now list a number of conditions that collectively make the inequality (3) true.

Conditions

C1. $C_{\{0\}}$ =a log N, a is a positive constant and N is the size of the file represented by node n_0.

C2. $a \log \gamma N \le C_i \le a \log \delta N$, $a, \delta, \gamma > 0$, $1 \le i \le m$

C3. $f_i \ge \alpha f_0$, $\alpha > 0$, $1 \le i \le m$

C4. $fx_i \le \beta f_0$, $\beta > 0$, $1 \le i \le m$

Please note that the conditions C1 and C2 simply state that all nodes have the same query cost function (i.e., logarithmic cost). The condition C2 also states that the ratio of file sizes between the child nodes and the father node ranges between γ and δ. The condition C3 gives a lower limit on the frequency ratio between the father node and the child nodes. The condition C4 gives an upper limit on the frequency ratio between the father node and the edges between the father node and the child nodes.

From the inequalities (1) and (2), we have

$$C_{\{0\}}\,(f_0 + \sum_{i=0}^{m} fx_i) + C_{\{j\}}\,(f_j + fx_j) + C_{\{k\}}\,(f_k + fx_k) \le C_{\{0,j\}}\,(f_0 + f_j + \sum_{i=0}^{m} fx_i) + C_{\{0,k\}}\,(f_0 + f_k + \sum_{i=0}^{m} fx_i) - C_{\{0\}}\,(f_0 + \sum_{i=0}^{m} fx_i) \quad (4)$$

By using the inequality (4), the inequality (3) is true if the inequality

$$C_{\{0,j\}}\,(f_0 + f_j + \sum_{i=0}^{m} fx_i) + C_{\{0,k\}}\,(f_0 + f_k + \sum_{i=0}^{m} fx_i) - C_{\{0\}}\,(f_0 + \sum_{i=0}^{m} fx_i) \le C_{\{0,j,k\}}\,(f_0 + f_j + f_k + \sum_{i=0}^{m} fx_i) \quad (5)$$

is true. By utilizing conditions C1, C2, C3 and C4, one can show [HafO88] that the inequality (5) is true if

$$\frac{\log((1+\delta)^2/(1+2\gamma))}{\log((1+2\gamma)/(1+\delta))^2} \le \frac{\alpha}{1+m\beta} \quad (6)$$

Thus, given the inequality (6) and the conditions C1, C2, C3 and C4, the inequality (3) always holds. This discussion is generalized in the lemma given below.

Lemma 6: Consider a two-level subtree ST, with the node set SN, $E_{\{0\}} \le E_{\{0,i\}}$; $n_i \in$ SN. Assume that the conditions C1, C2, C3 and C4 are satisfied. Then the inequality $E_{\{0\}} \le E_{K_l}$, $K_l = \{ i_0, i_1, \dots, i_l \}$, $n_{i_j} \in$ SN, $0 \le j \le l$, $1 \le l \le n$, $n = |SN| - 1$, is true if

$$\frac{\log((1+\delta)^l/(1+l\gamma))}{\log((1+l\gamma)/(1+\delta))^l} \le \frac{\alpha}{1+m\beta}$$

Lemma 7: Let $C_{n,q}$ = a log N, a is a positive constant, n $\in$ SN, ST be a two-level subtree, $E_{\{0\}} \le E_{\{0,i\}}$, $n_i \in$ SN. Assume conditions C1, C2, C3 and C4 hold. If the inequality (6) holds then the optimum subtree of ST is ST itself.

3.3.1. Evaluating the Cases in Which the Sufficient Condition Holds

In this section, we discuss experimentally the effect of applying the GREEDY-MERGE algorithm to the scheme tree T when there is only one type of queries involved in the database system workload.

According to lemmas 4 and 5, the GREEDY-MERGE algorithm produces the optimum scheme tree when the query types involved in the database workload have query (processing) costs a, a N^k and N g(N); $a > 0$, $K \ge 1$, N is the size of the file involved in the query and g(N) is a monotonically increasing function of N. For the logarithmic query processing cost, lemma 7 gives a sufficient (but not necessary) condition for which the GREEDY-MERGE algorithm produces optimum two-level subtrees. In this section, we discuss the effect of the database workload on the inequality (6) in lemma 7.

Assume the two-level subtree ST in figure 4 is the subtree that results after applying the GREEDY-MERGE algorithm. The inequality (6) gives a sufficient condition for the optimum two-level subtree ST. We now observe the behaviour of the inequality (6) when the parameters α, β, δ and γ change. For m=5, the number of optimum subtrees satisfying the inequality (6) increases when α and γ increase. On the other hand, the number of optimum subtrees satisfying the inequality (6) decreases when β and δ increase. Figure 6 contains for different values of α, β, δ and γ, the variations in the ratio opt / total where opt is the number of optimum subtrees that satisfy the inequality (6), and total is the total number of subtrees.

Figure 6(a) shows that, all the cases with $0.1 \le \beta \le 0.5$, $0.5 \le \delta \le 2$ and $0.5 \le \gamma \le \delta$ satisfy the inequality (6) and are also optimum when $\alpha \ge 1.8$. Figure 6(b) shows that for $0.5 \le \alpha \le 5$, $0.5 \le \delta \le 2$ and $0.5 \le \gamma \le \delta$, all the cases satisfy the inequality (6) and are optimum if $\beta < 0.2$. In figure 6(c), all the cases are optimum for $\delta < 0.5$ when $0.5 \le \alpha \le 5$, $0.1 \le \beta \le 0.5$. Finally, figure 6(d) shows that all the cases are optimum for $\gamma > 3.0$ when $0.5 \le \alpha \le 5$ and $0.1 \le \beta \le 0.5$.

To get a better insight into the performance of the GREEDY-MERGE algorithm, we clarify some important factors which show the advantages of our greedy algorithm. For the two-level subtree shown in figure 4, the GREEDY-MERGE algorithm always chooses to merge the child node n_i to its father node n_0 as long as it gives the smallest scheme subtree cost. Assume the two nodes n_i and n_j, $1 \le i,j \le r$, are qualified to be merged to node n_0, i.e.,

$$C_{\{0\}}(f_0 + \sum_{k=0}^{r} fx_k) + C_{\{i\}}(f_i + fx_i) >$$
$$C_{\{0,i\}}(f_0 + f_i + \sum_{k=0}^{r} fx_k) \text{ and}$$
$$C_{\{0\}}(f_0 + \sum_{k=0}^{r} fx_k) + C_{\{j\}}(f_j + fx_j) >$$
$$C_{\{0,j\}}(f_0 + f_j + \sum_{k=0}^{r} fx_k)$$

Assume also that the GREEDY-MERGE algorithm chooses node n_j to be merged to node n_0, then

$$C_{\{0,j\}}(f_0 + f_j + \sum_{k=0}^{r} fx_k) - C_{\{j\}}(f_j + fx_j) \leq$$
$$C_{\{0,i\}}(f_0 + f_i + \sum_{k=0}^{r} fx_k) - C_{\{i\}}(f_i + fx_i)$$

which is equivalent to

$$C_{\{0,j\}}(f_0 + \sum_{k=0}^{r} fx_k) + (C_{\{0,j\}} - C_{\{j\}})f_j - C_{\{j\}}fx_j \leq$$
$$C_{\{0,i\}}(f_0 + \sum_{k=0}^{r} fx_k) + (C_{\{0,i\}} - C_{\{i\}})f_i - C_{\{i\}}fx_i. \qquad (7)$$

For $f_k = \alpha_k f_0$, $fx_k = \beta_k f_0$ and $C_{\{k\}} = \log(1 + \gamma_k)\, C_{\{0\}}$, k = 1, 2,. . . , r, the inequality (7) is always satisfied since the node n_j satisfies at least one of the following conditions,

(a) $f_j \leq f_i$ or $\alpha_j > \alpha_i$
(b) $fx_j \geq fx_i$ or $\beta_j < \beta_i$
(c) $C_{\{0,j\}} \leq C_{\{0,i\}}$, which means $C_{\{j\}} \leq C_{\{i\}}$ or $\gamma_j < \gamma_i$.

Conditions (a), (b) and (c) show that the resulting two-level subtree produced by the GREEDY-MERGE algorithm should have either large α values or small β values or large γ values or any of the above. With the support of the results established from figure 6, the two-level subtree ST obtained by the GREEDY-MERGE algorithm approaches the optimum when α or γ increases, or β decreases. As a result, the GREEDY-MERGE algorithm almost always locates a "near" optimum two-level subtree.

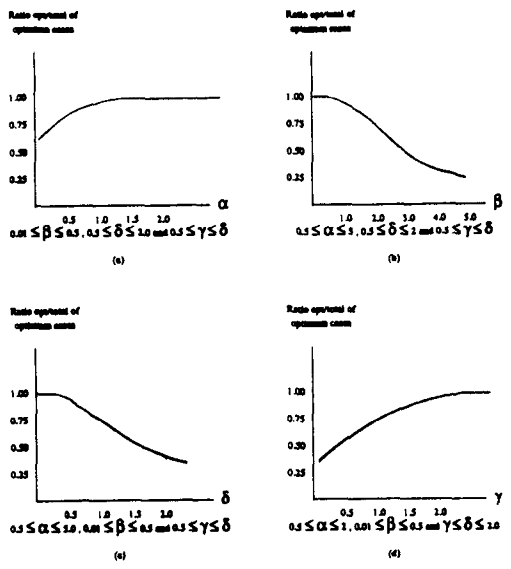


Figure 6. The System Workload Effect on the Number of Optimal Cases.

4. Preliminary Experimental Results

In this section, we compare the scheme trees obtained by applying the GREEDY-MERGE algorithm with the optimum scheme tree obtained by exhaustive enumeration for only one specific scheme tree. More experiments are presently being conducted for different scheme trees.

The results of applying the GREEDY-MERGE algorithm on the scheme tree T in figure 7 are shown in tables 1 - 5. We compare the scheme trees obtained by the GREEDY-MERGE algorithm and the optimum scheme trees obtained by exhaustive enumeration over all possible scheme trees. Only those cases that the GREEDY-MERGE algorithm produces non-optimum scheme trees are considered. In tables 1 - 5, we calculate *the percentage of error*, defined as the ratio e = ((a - b) / b)*100 where a is the cost of the resulting scheme tree obtained by the GREEDY-MERGE algorithm, and b is the cost of the optimum scheme tree. The frequencies of nodes and edges range from 1 to 5, except in tables 2 and 3 where the frequencies of nodes are assigned according to the node levels. When the frequencies range from 1 to 5, the results stay the same when the frequencies range from 1*A to 5*A with step A, A > 0, (that is, when A=10, the range is 10 to 50 with step 10). The sizes of the files range from 64 to 32,768 blocks, except in tables 4 and 5 where the sizes of files are assigned according to the node levels.

Tables 1 - 5 show two significant results. First, the number of cases that the GREEDY-MERGE algorithm does not locate the optimum scheme tree, is very small compared to the total number of cases.

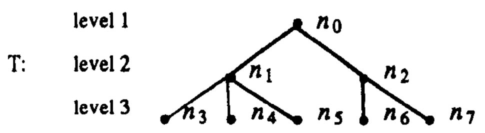


Figure 7

n	fx/f	% error
64	2/3	1.100
	3/5	0.555
128	1/2	0.632
256	2/5	0.349
512	1/3	0.293
1024	1/5	0.299
2048	1/3	0.389
	1/4	0.198
8192	1/4	0.337
	1/5	0.142
16384	1/4	0.213
32768	1/5	0.248

all nodes have the same frequency f
all edges have the same frequency fx
all files have the same size n blocks

Table 1.

fx	n	% error
1	16384	0.013
2	512	0.040
3	2048	0.047
	8192	0.087
	16384	0.136
	32768	0.196
4	1024	0.065
	2048	0.182
	4096	0.279
	16384	0.275
	32768	0.346
5	1024	0.130
	2048	0.266
	4096	0.379
	16384	0.367

all edges have the same frequency fx
all files have the same size n blocks
all nodes have level-wise frequencies
(level 3 has frequency = 100, level 2 has frequency = 10, . . . etc.)

Table 2.

fx	n	% error
1	4096	0.133
	8192	0.406
	16384	0.632
	32768	0.531
2	256	0.510
	512	1.182
	2048	0.999
	4096	1.432
	8192	1.139
	32768	1.235
3	512	0.404
	1024	1.074
	2048	1.708
	16384	1.473
	32768	1.836
4	512	0.391
	1024	0.918
	2048	1.451
	4096	2.370
	16384	1.880
	32768	1.548
5	512	0.123
	1024	0.671
	2048	1.148
	4096	1.843
	8192	2.441
	32768	2.024

all edges have the same frequency fx
all files have the same size n blocks
all nodes have level-wise frequencies
(level 3 has frequency = 1, level 2 has frequency = 10, . . . etc.)

Table 3.

f	fx	% error
1	1	1.880
2	1	1.358
	2	0.220
3	2	3.866
	3	1.326
	4	0.610
4	3	4.257
	4	2.423
	5	1.094
5	4	4.359
	5	2.962

all nodes have the same frequency f
all edges have the same frequency fx
all files have level-wise sizes
(level 3 has size = 256 blocks, level 2 has size =128 blocks, . . . etc.)

Table 4.

f	fx	% error
2	2	2.786
	3	0.849
	4	0.512
3	3	4.462
	4	3.041
	5	1.896
4	4	5.557
	5	4.288
5	5	5.741

all nodes has the same frequency f
all edges have the same frequency fx
all files have level-wise sizes
(level 3 has size = 64 blocks, level 2 has size =128 blocks, . . . etc.)

Table 5.

Second, the percentage of error e is significantly small.

In table 1, the percentage of error e decreases when the ratio between fx and f decreases. For example when n=64 and fx/f = 2/3 the percentage of error is 1.1%, while at the same value of n=64 and fx/f = 3/5 the percentage of error is 0.555%. Tables 2 and 3 show that the percentage of error decreases when the ratio between the child frequency and the father frequency is large. For fx =1 and n=16384, when the ratio between the child frequency and the father frequency is large (e.g., child frequency is 10 and father frequncy is 1), table 2 shows that the percentage of error is 0.013%, while in table 3, when the ratio of frequencies is small, (e.g., child frequency is 10 and father frequency is 100), the percentage of error is 0.632%. In tables 4 and 5, the percentage of error decreases when the ratio between the child's file size and the father's file size is large. For f =2 and fx =2, when the ratio is large, table 4 shows that the percentage of error is 0.220%, while in table 5, when the ratio is small, the percentage of error is 2.786%.

5. References

[AbiB84] Abiteboul, S. and Bidoit, N., "Non First Normal Relations to Represent Hierarchically Organized Data", Proc., *the 3rd ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, Apr. 1984.

[CDRS86] Carey, M.J., DeWitt, D.J., Richardson, J.E. and Shekita, E., "Object and File Management in the EXODUS Extensible Database System", *Int. Conf. on VLDB*, Aug. 1986.

[CopK85] Copeland, G. and Khoshafian, S., "A Decomposed Storage Model", *ACM SIGMOD Int. Conf. on Management of Data*, May 1985.

[Date87] Date, C., *An Introduction to Database Systems* (4th ed.), Addison-Wesley, 1987.

[DesV88] Deshpande, A. and Van Gucht, D., "An Implementation for Nested Relational Databases", Technical Report, Computer Science Dept., Indiana University, Feb. 1988.

[DKAB86] Dadam, P., Kuespert, K., Andersen, F., Blanken, H., Erbe, R., Guenauer, J., Lum, V., Pistor, P. and Walch, G., "A DBMS Prototype to Support Extended NF^2 Relations: An Integrated View on Flat Tables and Hierarchies", Proc. *ACM SIGMOD Int. Conf. on the Management of Data*, May 1986.

[HafO88] Hafez, A. and Ozsoyoglu, G.,"The Partial Normalized Storage Model of Nested Relations", Technical Report, Department of Computer Science, Case Western Reserve University, 1988.

[HamN78] Hammer, M. and Niamir, B., "A Heuristic Approach to Attribute Partitioning", Proc. *ACM SIGMOD Int. Conf. on Management of Data*, May 1979.

[HofS75] Hoffer, J.A. and Severance, D.G.,"The Use of Cluster Analysis in Physical Database Design", Proc., *2nd Int. Conf. on VLDB*, 1975.

[KCJB87] Khoshafian, S., Copeland, G., Jagodits, T., Boral, H. and Valduriez, P., "A Query Processing Strategy for the Decomposed Storage Model", *3rd Int. Conf. on Data Engineering*, Feb. 1987.

[KiCB87] Kim, W., Chou, H. and Banerjee, J., "Operations and Implementation of Complex Objects", *3rd Int. Conf. on Data Engineering*, Feb. 1987.

[McGe77] McGee, W.C., "The Information Management System IMS/VS Part 1: General Structure and Operation", *IBM Syst. J., Vol. 16, No. 2*, 1977.

[NCWJ84] Navathe, S., Ceri, S.,Wiederhold, G. and Jinglie, D., "Vertical Partitioning Algorithms for Database Design", *ACM Trans. on Database Systems, Vol.8, No.4*, Dec. 1984.

[Olle71] Olle, T.W., "Introduction to 'Feature Analysis of Generalized Data Base Management Systems'", *CACM, Vol.14, No.5*, May 1971.

[RoTK82] Rotem, D., Tompa, F. and Kirkpatrick, D., "Foundations for Multiple Design by application Partitioning", Proc., *ACM Symposium on Principles of Database Systems*, Mar. 1982.

[StoR86] Stonebraker, M. and Rowe, L.A., "The Design of POSTGRES", Proc., *ACM SIGMOD Int. Conf. on Management of Data*, May 1986.

[VaKC86] Valduriez, P., Khoshafian, S. and Copeland, G., "Implementation Techniques of Complex Objects", *Int. Conf. on VLDB*, Aug. 1986.

[ValB86] Valduriez, P. and Boral, H., "Evaluation of Recursive Queries Using Join Indices", *Proc. of First Int. Conf. on Expert Systems*, Apr. 1986.

[vald87] Valduriez, P., "Join Indices", ACM Trans. on Database Systems, Vol.12, No. 2, June 1987.

[Wied83] Wiederhold, G., *Database Design* (2nd ed.), McGraw-Hill, 1983.