A Methodology for Vertically Partitioning in a Multi-Relation Database Environment

Narasimhaiah Gorla
Professor of MIS
American University of Sharjah, UAE
ngorla@aus.edu, n_gorla@yahoo.com

ABSTRACT

Vertical partitioning, in which attributes of a relation are assigned to partitions, is aimed at improving database performance. We extend previous research that is based on a single relation to multi-relation database environment, by including referential integrity constraints, access time based heuristic, and a comprehensive cost model that considers most transaction types including updates and joins. The algorithm was applied to a real-world insurance CLAIMS database. Simulation experiments were conducted and the results show a performance improvement of 36% to 65% over unpartitioned case. Application of our method for small databases resulted in partitioning schemes that are comparable to optimal.

Keywords*:* Vertical partitioning, Database performance, Referential integrity constraints, Multi-relation databases.

1. INTRODUCTION

Data volumes are increasing at an astonishing rate in the commercial world due to increase in number and complexity of transactions. In spite of advances in computer technology, data access performance still remains a critical issue in information system. Vertical partitioning is a physical database design technique that is aimed at improving the access performance of user transactions. In vertical partitioning, a relation is split into a set of smaller physical files, each with a subset of the attributes of the original relation. The rationale is that normally database transactions require access only to subset of the attributes. Thus, if we can split the relation into sub files that closely match the requirements of user transactions, the access time for transactions reduces significantly.

Several researchers have made significant contributions for over two decades in the area of vertical partitioning [5,8,17]. Research contributions in vertical partitioning have been made in the areas of, for example, attribute oriented approach [17], transaction oriented approach [7], combined vertical partitioning and access methods [8,28], distributed databases [5], and object-oriented databases [12,14]. To our knowledge, previous research has dealt with the vertical partitioning problem by considering single relation only and ignored the impact of database operations in a multi-relational context. In a logical database schema, each relation is connected with one or more relations through primary/foreign key links and data integrity is achieved through enforcement of the referential integrity constraints. Since update transactions tend to violate these referential integrity constraints, the relations and attributes are accessed and/or updates performed as necessary in order to maintain database integrity. These integrity enforcing operations affect the "best" attribute partitioning scheme and ignoring these can result in "suboptimal" partitioning solution. The effect is even more prominent in volatile databases where the frequency of update operations is high. Our methodology explicitly considers these influences in determining the "best" fragmentation scheme. Furthermore, unlike in most previous research, we use access time of transactions in determining and evaluating best partitioning schemes. In this research, we extend one-relation based vertical partitioning to multi-relation environment by including referential integrity constraints, modeling a comprehensive cost function, considering system/disk access characteristics, and using differential access times of various transaction types.

The objective of this research is to provide a general approach for vertically fragmenting relations in a multi-relation environment. Since the problem is computationally intractable, we use a heuristic procedure to solve the problem using a 2-attribute affinity index and a 2-step clustering algorithm. The application of our methodology on small problems yielded optimal solutions obtained by exhaustive enumeration. We ran simulation experiments under varying updates and join operations in order to validate our proposed method. We also compared our results with the solutions obtained by two previous studies.

The organization of the paper is as follows. Section 2 provides a description of previous works on vertical partitioning in relational databases. In section 3, we describe the proposed partitioning method for a multi-relation environment. Section 4 has database performance model that is developed in this research, which is a comprehensive access-time formula. Section 5 has the application of the procedure on an insurance company's CLAIMS database. Section 6 is regarding evaluation of our solution procedure, including results of simulation experiments, comparison with exhaustive enumeration, and comparison with solution of previous researchers. Section 7 contains conclusions and directions for future research.

2. PREVIOUS RESEARCH

Because of the criticality of the database performance, several researchers have contributed enormously to vertical partitioning.. Database partitioning has been applied in centralized relational databases [4,8,17,25,28], distributed databases [2,5,8,19,22,26], Data Warehouse Design [10,13,18], and Object-Oriented Database design [12,14].

Hoffer and Severance [17] consider the vertical partitioning problem by applying bond energy algorithm on similarity of attributes, which are based on access patterns of transactions. Their work was extended by Navathe, Ceri, Widerhold, and Dou [23] by presenting vertical partitioning algorithms for three contexts: a data base stored on devices of a single type; in different memory levels; and a distributed database. They used affinity between attributes for partitioning, which is based on number of disk accesses. An alternate graphical approach was proposed by Navathe and Ra [24]. Cornell and Yu [8] used an optimal binary-partitioning algorithm to obtain vertical partitioning, which is iteratively applied to obtain more partitions. The study uses number of

accesses to evaluate partitions. Chu and Ieong [7] develop a transaction-based approach to vertical partitioning, in which transaction rather than attribute is used as the unit of analysis. Song and Gorla [28] used genetic algorithms to obtain solutions simultaneously for vertical partitions and access paths for those partitions. They also used the number of disk accesses as the partitioning evaluation criterion. Cheng, Lee, and Wong [5]use genetic search-based clustering algorithm based on traveling salesman problem to obtain vertical partitions in distributed databases. With reference to object-oriented database design, Gorla [14]used genetic algorithm to determine the instance variables that should be stored in each class/ subclass in a subclass hierarchy, so that the total cost of database operations is minimized. More recently, Ailamaki et al [1]proposed Partition Attributes Across (PAX) model by improving cache performance, while Ramamurthy et al [27] proposed fractured mirrors partitioning scheme based on Decomposition Storage Model and N-ary Storage Model. Fung, Karlapalem, and Li [12] analyze vertical partitioning of classes/ subclasses for class composition hierarchy and subclass hierarchy and develop the associated cost functions for query processing under the cases of large memory and small memory availability. Ng et al [25]proposed a combined vertical partitioning and tuple clustering using genetic algorithm.

We extend previous research of single relation cases by providing a procedure for vertical partitioning of relations in a multi-relation database environment. An important characteristic that distinguishes multi-relation schema from single relation case is referential integrity constraints enforcement due to update transactions. Our approach makes use of a 2-attribute affinity, as used in previous studies of Navathe et al [23] and Cornell and Yu [8]. However, we differ from their approach in that our attribute affinity metric is based on differential access times of transactions rather than number of disk accesses. There is a substantial difference between these two methods of evaluation, since fetching an additional block of records from disk in a sequential scan takes much less time than fetching an arbitrary block randomly, which takes even less time than the time for inserting/deleting a record. Our access time computations are baaed on disk I-O service times. Furthermore, our algorithm is similar to "hill climbing" [16] in that our algorithm groups attributes such that the objective function keeps increasing; our approach differs theirs in that we have two steps to our algorithm – grouping and verifying. Thus, we extend previous research on vertical partitioning by including referential integrity constraints and join transactions, and by using a comprehensive cost function to evaluate fragmentation scheme that is based on access time rather than count of accesses. Physical database design provides truly optimal performance when the design is made to fit specific disk characteristics and is only optimal on the given hardware architecture [20]. Our proposed methodology includes disk access characteristics as part of our attribute affinity measure.

3. VERTICAL PARTITIONG PROCEDURE

The vertical partitioning problem in a multi-relation environment is stated as follows: *Given a relational schema, the retrieval/update/join transactions on the schema, the referential integrity constraints among relations, and the disk access parameters, the objective is to determine stored fragments for each relation, which results in the minimum total database access costs.* The partitioning problem is computationally complex. Consider a relational schema with N relations, with A_i attributes for relation i. A relation with A attributes can be partitioned in B(A) different ways [16], where B(A) is the A^{th} Bell number (for A=30, B(A) = 10^{15}). Using exhaustive enumeration, the number of possible fragmentations for the N-relation schema is approximately $B(A_1)B(A_2)$ ... $B(A_N)$. Yu et al [34]find out that the number of attributes for base tables and views in a typical relational environment are 18 and 41 respectively. Even if we consider a small schema of 10 relations with 15 attributes per relation, the number of possible fragments is approximately $(10^9)^{10}$ =10^{90}. Since the problem is intractable, solving large problems requires the use of heuristic techniques. Our procedure consists of three steps. First, database transactions on the logical schema are transformed into transactions on individual relations. Second, an attribute grouping benefit index (AGBI) is computed. Third, a clustering algorithm using AGBI is applied to derive effective fragments.

Transaction Analysis

Single relation transactions are of two types: retrievals and updates; retrievals can be sequential scans or random retrievals; updates can be inserts, deletes, or modifications. We use 3-tuple transaction mix: (updates: single relation retrievals: joins). While the single relation retrievals cause no problems, the update transactions may violate the referential integrity constraints. For example, inserting a tuple can violate integrity if the value of the foreign key does not exist in the referenced relation. Deletion operation can violate integrity if the related foreign keys reference tuple being deleted. One of the options to preserve referential integrity is to delete tuples that reference the tuple being deleted; we avoid cascade deletes by setting the foreign keys to null. Modifying a primary key value is similar to adding and deleting tuples.

A referential integrity constraint between relation *rn* and referenced relation *rf* implies that a foreign key value in *rn* should match primary key value in *rf*. Thus, a transaction that tries to insert a record in relation *rn* will generate a random transaction on relation *rf* to ensure that there is a matching primary key value. A delete transaction on relation *rn* does not generate any additional transactions. Similarly, a delete transaction on relation *rf* will generate additional delete transactions on relation *rn*, which will delete all the records in relation *rn* that corresponds to the primary key of the record in *rf*. On the other hand, an insert transaction on relation *rf* will not generate any additional transactions.

Join transactions can be processed using any of the four methods [11]: inner-outer loop, sort-merge, and using an access structures such as index-join and hash-join. We assume that join transactions use some access structure (e.g. index) on join-attribute to retrieve joined records. Furthermore, we ignore costs associated with access structure. A join transaction such as "*Select R1.a2, R1.a3, R2.b3 from (R1 left outer join R2) on R1.a1=R2.b3*" is executed by sequentially retrieving R1 and retrieving matching tuple(s) from R2 for each tuple of R1. Thus we convert the join transactions into a sequential retrieval transaction on R1 (i.e. *Select a2, a3 from R1*) and a random retrieval transaction on R2 (i.e., *Select b3 from R2 where b3=a1*).

Attribute Grouping Benefit Index (AGBI)

Notation: Let c (r) be the cardinality of relation

r, $L_{i(r)}$ be the length of attribute i of relation r, and B the block size. The disk access parameters are t_l (latency time), t_s (seek time), t_{bt} (block transfer time), t_{br} (block read time: $t_l + t_{bt}$), t_{ar} (time to access block and read: $t_s + t_{br}$), and t_{rw} (rewrite time: $2t_l$). The rewrite time involves one latency time to locate the record and another latency time to write it [31]. Also let transaction (k) be of the type q (s: sequential retrieval, r: random retrieval, d: delete, m: modify, and i: insert).

In order to compute AGBI based on 2-attribute grouping, transactions are categorized with reference to attributes i and j into three types: i) transactions that need access (for read or for write) to attribute i but not j, ii) transactions that need access to attribute j but not i, and iii) transactions that need access to attributes i and j. Transactions that do not access either i or j are not considered for AGBI computation. The other notations are as follows:

$F^{qk}_{y(r)}$ = Frequency of k^{th} transaction of type q accessing y^{th} attribute of relation r, where y ∈ (i, j, ij)
/* i = attribute i only, j = j only; ij = both i and j */

$F^{q}_{t(r)}$ = Total frequency of all transactions of type q accessing attributes i and/or j in relation r

$$= \sum_{k} (F^{qk}_{i(r)} + F^{qk}_{j(r)} + F^{qk}_{ij(r)})$$

AVR^q_r = Average no. of records accessed by all transactions of type q in relation r
=(selectivity)(cardinality)(frequency)/(total frequency)

$$= \sum_{k} \sum_{y\in(i,j,ij)} (Sel^{qk}_{y(r)} . c(r) * F^{qk}_{y(r)}) \Big/ \sum_{k} \sum_{y\in(i,j,ij)} F^{qk}_{y(r)}$$

for q ∈ (s,r,m)

$$= \sum_{k} (Sel^{qk}_{i(r)} . c(r) . F^{qk}_{T(r)}) \Big/ \sum_{k} F^{qk}_{T(r)}, \text{ for } q \in (i,d)$$

In the expression for AVR, the numerator is total number of records accessed (selectivity x cardinality) weighted by transaction frequency and the denominator is the total frequency of those transactions. In case of sequential retrievals, random retrievals, and modify transactions, the transactions that access attributes i and/or j need to be considered, since these transactions access only a subset of attributes. In case of insert and delete transactions, since they affect all the attributes in the relation, all transactions that access the relation are considered to be accessing attributes i and j. The AVR expression is used in the computation of AGBI.

AGBI Computation

Access cost and access time are used interchangeably in this research. AGBI, an attribute-affinity measure, represents the benefit in terms of access time obtainable by storing the two attributes in one fragment compared to storing them in separate fragments. AGBI is calculated for each pair of attributes i and j, considering the two cases: i) when attributes i and j are stored together as one fragment and ii) when attributes i and j are stored in separate fragments. AGBI is computed for each transaction type q ∈ (s, r, i, m, d), which are then totaled for all transaction types.

$COST\text{-}COM^{q}_{ij(r)}$ = Database operating cost with type q transactions when attributes i and j of relation r are stored in separate fragments.

$COST\text{-}COM^{q}_{ij(r)}$ = Database operating cost with type q transactions when attributes i and j of relation r are stored in the same fragment.

$AGBI^{q}_{ij(r)}$ = Attribute Grouping Benefit Index

$$= COST\text{-}SEP^{q}_{ij(r)} - COST\text{-}COM^{q}_{ij(r)}$$

The AGBI computations for sequential retrieval, random retrieval, insertion, deletion, and modify transactions are shown below.

1. Sequential Retrievals:

The disk access time is the time to transfer all blocks of the fragment from disk to buffer and is computed as (time to transfer a block of records) x (transaction frequency) x (cardinality) x (fragment length). When attributes i and j are stored in the same fragment, the record length of the fragment is length of attribute i ($L_{i(r)}$) + length of attribute j ($L_{j(r)}$). Thus,

$COST\text{-}COM^{s}_{ij(r)} = t_{br} . F^{s}_{t(r)} . c(r) . (L_{i(r)} + L_{j(r)}) /B.$

When attributes i and j are stored in separate fragments, the access times need to be computed separately for transactions that access attribute i only, transactions that access attribute j only, and transactions that access attributes i and j. In each case, the access time is calculated as (time to transfer a block of records) x (frequency of transactions) x (cardinality) x (fragment length).

$COST\text{-}SEP^{s}_{ij(r)} = t_{br} . (c(r) . L_{i(r)} . F^{s}_{i(r)} /B + c(r) . L_{j(r)} . F^{s}_{j(r)} /B + c(r) . (L_{i(r)} + L_{j(r)}) . F^{s}_{ij(r)} /B)$

Thus, $AGBI^{s}_{ij(r)} = -t_{br} . c(r) . (F^{s}_{j(r)}.L_{i(r)} + F^{s}_{i(r)}.L_{j(r)})/B$

The negative sign of AGBI indicates it is not beneficial to group attributes together in case of sequential retrieval transactions. Here we ignore CPU time needed internally to combine the fragments, since disk I-O times dominate the internal CPU times. In a database environment, where transactions are predominantly of sequential in nature, it is more efficient to have highly fragmented relations.

2. Random Retrievals:

The database access time for random retrieval transactions is disk access and read time (t_{ar}) x frequency of random transactions x number of blocks accessed per transaction as per (Cardenas, 1975).

$$COST\text{-}COM^{r}_{ij(r)} = t_{ar} . F^{r}_{t(r)} .(1-(1-1/BLKS)^{AVR^{r}_{r}.BLKS})$$
$$= t_{ar} . F^{r}_{t(r)} . AVR^{r}_{r} \quad \text{(first approximation)}$$

where $BLKS = c(r).(L_{i(r)}+L_{j(r)})/B$

(AVR^{r}_{r} is the no. of records accessed by random transactions requiring attributes i and/or j in relation $_r$).

$COST\text{-}SEP^{r}_{ij(r)} = t_{ar} (F^{r}_{i(r)}.AVR^{r}_{i(r)} + F^{r}_{j(r)}.AVR^{r}_{j(r)} + 2.F^{r}_{ij(r)}.AVR^{r}_{ij(r)})$

In the above expression, $AVR^{r}_{i(r)}$, $AVR^{r}_{j(r)}$, and $AVR^{r}_{ij(r)}$ are the number of records accessed by random retrievals requiring attribute i only, requiring attribute j only, and requiring both i and j, respectively. The factor 2 is applied to the last term in COST-SEP because when attributes i and j are stored in different fragments, transactions using both attributes i and j have to access both fragments.

$AGBI^r_{ij(r)} = t_{ar} . F^r_{ij(r)} . AVR^r_{ij(r)}$

The positive AGBI in the above expression implies that when predominantly random transactions exist, it is more efficient not to fragment the relations. As the access time for random transactions is proportional to number of records to be accessed (as in formulae above), more access time may result with highly fragmented relation.

3. Insert Transactions:
The time to insert a record depends on (time to read and write the record) * (number of records to be inserted). The 2 in COST-SEP is because insertion has to be performed in both the fragments. Here we assume random record insertion, thus there may be a write required for each record to be inserted, since there is little chance that subsequent records to be inserted falls into the same block.

$COST\text{-}COM^i_{ij(r)} = F^i_{t(r)} .(t_{ar} + t_{rw}).AVR^i_r$
$COST\text{-}SEP^i_{ij(r)} = F^i_{t(r)} . 2.(t_{ar}+t_{rw}).AVR^i_r$
$AGBI^i_{ij(r)} = F^i_{t(r)} . (t_{ar}+t_{rw}).AVR^i_r$

4. Deletion Transactions:
The time computation for deletion is similar to record insertion time described above.
$AGBI^d_{ij(r)} = F^d_{t(r)} . (t_{ar}+t_{rw}).AVR^d_r$

5. Modify Transactions: The time is calculated similar to the above.

$COST\text{-}COM^m_{ij(r)} = F^m_{t(r)} . (t_{ar}+t_{rw}).AVR^m_r$
$COST\text{-}SEP^m_{ij(r)} = F^m_{t(r)} . 2.(t_{ar}+t_{rw}).AVR^m_r$
$AGBI^m_{ij(r)} = F^m_{ij(r)} . (t_{ar}+t_{rw}).AVR^m_r$

The Total Grouping Benefit Index:

$$GBI^T_{ij(r)} = \sum_{q \in (s,r,i,d,m)} AGBI^q_{ij(r)}$$

Clustering Algorithm
The GBI_{ij} computed as above are entered into an n*n matrix, where n is the number of attributes in a relation. The algorithm (Figure 1) uses only positive GBI_{ij} because only positive ones contribute towards maximizing *SchemaValue*, thus reducing execution time to half on the average. The *SchemaValue* is the total value of GBIs in the existing fragments and it is a heuristic measure of the merit of a specific fragmentation scheme. The algorithm ensures that *SchemaValue* increases as fragments are generated. The algorithm is applied on each relation using its GBI matrix. The algorithm has two steps: a grouping step where the attributes with positive GBIs are grouped into fragments and a regrouping step where a verification is made if attributes with positive GBIs are in the same fragment and reassignment of attributes is made as needed.
If we assume an average of n attributes per relation, the size of the GBI matrix is n(n-1)/2. Since we only use positive values, there are on the average n(n-1)/4 elements to be processed per relation. Considering regrouping step, the number of elements to be processed is n(n-1)/2. For a schema with m relations, the complexity of the proposed algorithm is $O(m*n^2)$ and that of exhaustive enumeration is $O(n^{m*n})$.

Figure 1. Clustering Algorithm for Fragmentation
Step 1: (Grouping Step)

1.1. IF all $GBI_{ij} > 0$,
THEN group all attributes in one fragment, EXIT
ELSEIF all $GBI_{ij} < 0$,
THEN keep each attribute in a separate fragment, EXIT.
Initialize a one-dimensional array B of size (=number of attributes)
1.2. Pick an element with the highest positive GBI_{ij};
Group attributes i and j in one fragment.
Mark GBI_{ij}, GBI_{ji}, B_i, B_j;
SchemaValue ← GBI_{ij}
1.3. Do until no more unmarked positive GBI_{ij} exists:
Pick an unmarked highest positive GBI_{ij}.
1.3.1 Case: Neither attribute i nor j is already assigned
Group i and j into a new fragment
SchemaValue ← $+GBI_{ij}$.
1.3.2 Case: attribute i is assigned and attribute j is unassigned
Compute *INCR* = ΣGBI_{kj}, $V_{k,}$ where k is an attribute in fragment f;
IF *INCR* >0,
THEN assign j to fragment f
SchemaValue ← *+INCR*;
ELSE
Repeat for each fragment f' (other than f)
Compute *INCR'* = ΣGBI_{kj}, $V_{k,}$ where k is an attribute in fragment f';
IF *INCR' > Max-INCR'*
THEN *Max-INCR'* ← *INCR';*
IF *Max-INCR' > 0*
THEN store j in the corresponding f'
SchemaValue ← + *Max-INCR'*.
ELSE store attribute j in a new fragment.
1.3.3 Mark GBI_{ij}, GBI_{ji}, B_i, B_j.

Step 2: (Regrouping step)
2.1. Unmark all GBI_{ij} and B_i.
2.2. Repeat for each unmarked highest $GBI_{ij} > 0$:
IF i and j are not in the same fragment, THEN
2.2.1 Compute Net GBI, if attribute j is moved to another fragment f
Repeat the above for each fragment
Let the maximum Net GBI be $incr_j$ and the corresponding fragment be f_j
2.2.2 Repeat step 2.2.1 for attribute i
Let the maximum Net GBI be $incr_i$ and the corresponding fragment be f_i
2.2.3 IF max ($incr_j$, $incr_i$) > 0,
Move attribute (j or i) to the corresponding fragment (f_j or $f_{i)}$
SchemaValue ← + max ($incr_j$, $incr_i$);
Mark GBI_{ij}.

4. PERFORMANCE MODELING

When tuples of relations are fragmented, there should be a mechanism to identify the fragments of a tuple. There are two methods to do this. One way of relating individual fragments of a tuple is by duplicating primary key in all the fragments or by using tuple-identifiers. This method involves additional processing for accessing the records through primary key or tuple identifier. While this arrangement is useful for both fixed-length and variable-length record files, this arrangement is essential for variable length records. This method is also beneficial if one needs to reconstruct the data record should there be corruption of data in the database provided the key or

identifier is unaffected. In the second method, which is less robust, the individual data fragments can be accessed exploring the relative positions of records. For sequential scan, there is no need to identify the corresponding fragments on an individual basis. A sequential scan on a relation is realized as sequential scans on individual fragments. This is as though several files are read simultaneously into their own buffers. However, a fragment design implementation module should keep track of which fragment block to read next.

Database Operating Cost

The database operating cost function consists of several terms. The first term is total access time for sequential transactions (fragment lengths x cardinality x frequency x block transfer time). The second term is the total access time for all random retrievals (number of accesses needed for each fragment x frequency x block read time). The other terms are join cost, insertion cost, deletion cost, and referential integrity maintenance costs for deletions and insertions. Several of the terms used in this formula are described in section 2. The access-cost function for a fragmentation scheme is shown in Appendix A.

5. ILLUSTRATIVE EXAMPLE

Insurance CLAIMS Database:

The above proposed procedure (hereafter called MRP - Multi-Relation Partitioning procedure) is applied to a small real-life insurance company's CLAIMS database (BUCLAIM.MDB) for illustration purposes. The database contains 7 relations (CLAIMS, IMPORT ERRORS, KDB ARCHIVE FILES, KDB CLOSED, KDB DATA, POTENTIAL CLAIMS, WELDISP) with attributes ranging from 3 to 42 and tuples from 6 to 3534. The relations of CLAIMS database and the referential integral constraints are given in Table 1 and Table 2, respectively.

Table 1. CLAIMS Database

Reln	Attr	Records	Attribute Lengths
1	30	2099	9 100 20 20 10 8 20 32 8 8 100 10 25 20 8 15 10 30 10 50 100 8 20 100 20 8 8 10 50 50
2	3	6	255 255 4
3	5	1313	3 10 10 10 36
4	42	1377	7 13 9 7 7 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 3 5 7 18 13 8 8 8 9 6
5	42	447	7 13 9 7 7 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 3 5 7 18 13 8 8 8 9 6
6	15	99	50 15 8 100 100 15 15 15 8 25 100 8 25 25 50
7	29	3534	255 255 255 255 255 4 255 255 8 8 255 255 255 255 8 255 8 255 255 255 255 255 255 255 255 255 255 255

Table 2. Referential Integrity Constraints

		referenced relation						
		1	2	3	4	5	6	7
	1	0	0	0	0	1	1	0
	2	0	0	0	0	0	0	0
relation	3	0	0	0	1	1	0	0
	4	1	0	1	0	1	0	0
	5	1	0	1	1	0	1	0
	6	1	0	0	1	0	0	0
	7	0	0	0	0	0	0	0

The referential integral constraint is denoted by a "1" in Table 2. The referential integrity constraints are utilized to generate additional random retrieval transactions and additional delete transactions on the relations that reference it. For example, a "1" in (relation 5, referenced relation 4) implies that an insert transaction on relation 5 should generate an additional random retrieval transaction on referenced relation 4, for verifying the primary key value is present in relation 4.

The database transactions are composed of 36 inserts, 24 deletes, 108 random retrievals, 84 sequential transactions, and 48 join transactions. Sequential transactions are business reports extracted from relational tables. The details of these transactions are as given in Table 3.

The transactions in Table 3 are applied to the CLAIMS database. Using the referential integrity matrix in Table 2, additional transactions are generated for insert and delete transactions. The join transactions are decomposed into transactions on individual relations. Tables 4 and 5 show the total transactions (both original and additionally generated) for relations #1 and #6, respectively. These two relations are shown for illustration purposes. *Gen Random* transaction (Table 4) is generated as a result of insert transaction (2^{nd} transaction in Table 3) on relation #5, in order to satisfy referential integrity constraint.

Table 3.
Transaction Set for CLAIMS Database

Trans Type	Relation --- Attribute List --	Freq	No of Access*	No of Attributes
INSERT	1	24	17	-
all				
INSERT	5	12	15	-
all				
DELETE	4	24	11	-
all				
RANDOM	1	12	178	5
12,8,15,28,22				
RANDOM	1	12	299	6
12,8,22,15,28,6				
RANDOM	1	12	472	5
1,2,16,17,34				
RANDOM	1	12	339	8
8,6,12,13,14,15,28,22				
RANDOM	1	12	27	8
8,22,6,28,15,3,2,4				
RANDOM	1	12	423	7
1,8,18,28,6,7,12				
RANDOM	6	12	19	2
10,6				
RANDOM	1	12	113	4
6,8,28,12				
RANDOM	1	12	399	11
22,5,6,7,8,11,12,14,15,21,23				
SEQUENCIAL	1	12	-	5
8,11,12,22,6				
SEQUENCIAL	1	12	-	5
12,8,22,15,28				
SEQUENCIAL	1	12	-	8
12,8,22,6,11,15,13,14				

SEQUENCIAL	1	12	-	8
12,8,22,6,11,15,13,14				
SEQUENCIAL	5	12	-	5
34,1,2,16,17				
SEQUENCIAL	1	12	-	8
12,8,7,6,22,11,14,15				
SEQUENCIAL	1	12	-	8
12,22,5,6,7,8,11,14				
JOIN		12		
Sequential	1		-	8
12,22,5,6,7,8,11,14				
random	5		213	5
5,1,2,16,17				
JOIN		12		
Sequential	4		-	2
2,34				
random	6		292	5
15,13,6				
JOIN		12		
random	1		85	5
12,7,8,11,14				
random	6		125	4
15,6				
JOIN		12		
Sequential	3		-	5
1,2,3,4,5				
random	5		105	4
34,2,16,40				
* number of records accessed each time a transaction is executed.				
** "-" for insert or delete operations indicate all attributes.				

Table 5 lists all transactions considered for Relation #6. Three additional transactions are generated on relation #6 due to update transactions in "Total Transactions" Table. For example, Gen RANDOM transaction (marked *) is generated because of insert transaction on relation #1; Gen RANDOM transaction (marked **) is generated because of insert transaction on relation #5; Gen DELETE transaction (marked ***) is generated due to delete transactions on relation #4.

Then a GBI matrix is created for each relation, using the time-cost formulae shown in section 3. The computed GBI Matrices for relations 1 and 6 are shown in Tables 6 and 7, respectively.

Table 4
Transaction Set for Relation #1

Transaction Type	Freq	Accesses	Attributes
INSERT	24	17	- -
Gen RANDOM	12	15	1 1
RANDOM	12	178	5 12,8,15,28,22
RANDOM	12	299	6 12,8,22,15,28,6
RANDOM	12	472	5 1,2,16,17,34
RANDOM	12	339	8 8,6,12,13,14,15,28,22
RANDOM	12	27	8 8,22,6,28,15,3,2,4
RANDOM	12	423	7 1,8,18,28,6,7,12
RANDOM	12	113	4 6,8,28,12
RANDOM	12	399	11 2,5,6,7,8,11,12,14,15,21,23
SEQUENCIAL	12	0	5 8,11,12,22,6
SEQUENCIAL	12	0	5 12,8,22,15,28
SEQUENCIAL	12	0	8 12,8,22,6,11,15,13,14
SEQUENCIAL	12	0	8 12,8,22,6,11,15,13,14
SEQUENCIAL	12	0	8 12,8,7,6,22,11,14,15
SEQUENCIAL	12	0	8 12,22,5,6,7,8,11,14
JOIN-SEQUEN	12	0	8 12,22,5,6,7,8,11,14
JOIN-RANDOM	12	85	5 12,7,8,11,14

Table 5
Transaction Set for Relation #6

Trans Type	Freq	Accs	Attibutes
Gen RANDOM*	24	17	1 1
Gen RANDOM**	12	15	1 1
Gen DELETE***	24	11	0
RANDOM	12	19	2 10,6
JOIN-RANDOM	12	292	5 15,13,6
JOIN-RANDOM	12	125	4 15,6

Table 6. GBI Matrix for Relation #1
(Part 1 of 4)

Attr	1	2	3	4	5	6	7	8
1	0	140	0	0	-4	113	119	111
2	140	0	8	8	-43	-122	-65	-144
3	0	8	0	8	-9	-18	-13	-22
4	0	8	8	0	-9	-18	-13	-22
5	-4	-43	-9	-9	0	109	116	107
6	113	-122	-18	-18	109	0	230	471
7	119	-65	-13	-13	116	230	0	251
8	111	-144	-22	-22	107	471	251	0
9	0	0	0	0	-3	-10	-5	-12
10	0	0	0	0	-3	-10	-5	-12
11	-12	-130	-26	-26	109	118	130	121
12	111	-152	-30	-30	107	463	251	543
13	-4	-43	-9	-9	-15	78	-25	73
14	-10	-109	-22	-22	111	214	134	235
15	-8	-79	-9	-9	106	308	101	362
16	140	140	0	0	-7	-20	-10	-23
17	140	140	0	0	-4	-13	-7	-15
18	125	0	0	0	-13	86	105	79
19	0	0	0	0	-4	-13	-7	-15
20	0	0	0	0	-22	-65	-33	-76
21	0	0	0	0	74	-12	53	-34
22	-14	-144	-22	-22	107	313	101	367
23	0	0	0	0	109	92	105	87
24	0	0	0	0	-43	-130	-65	-152
25	0	0	0	0	-9	-26	-13	-30
26	0	0	0	0	-3	-10	-5	-12
27	0	0	0	0	-3	-10	-5	-12
28	123	-14	4	4	-7	340	114	395
29	0	0	0	0	-22	-65	-33	-76
30	0	0	0	0	-22	-65	-33	-76

* The above numbers are obtained by dividing actual GBIs by 1000 and rounding

Table 6. GBI Matrix for Relation #1
(Part 2 of 4)

Attr	9	10	11	12	13	14	15	16
1	0	0	-12	111	-4	-10	-8	140
2	0	0	-130	-152	-43	-109	-79	140

3	0	0	-26	-30	-9	-22	-9	0
4	0	0	-26	-30	-9	-22	-9	0
5	-3	-3	109	107	-15	111	106	-7
6	-10	-10	118	463	78	214	308	-20
7	-5	-5	130	251	-25	134	101	-10
8	-12	-12	121	543	73	235	362	-23
9	0	0	-10	-12	-3	-9	-7	0
10	0	0	-10	-12	-3	-9	-7	0
11	-10	-10	0	121	-22	139	91	-20
12	-12	-12	121	0	73	235	354	-23
13	-3	-3	-22	73	0	84	89	-7
14	-9	-9	139	235	84	0	210	-16
15	-7	-7	91	354	89	210	0	-13
16	0	0	-20	-23	-7	-16	-13	0
17	0	0	-13	-15	-4	-11	-9	140
18	0	0	-39	79	-13	-33	-26	0
19	0	0	-13	-15	-4	-11	-9	0
20	0	0	-65	-76	-22	-54	-43	0
21	0	0	-12	-34	-43	9	31	0
22	-12	-12	96	359	73	209	362	-23
23	0	0	92	87	-9	96	101	0
24	0	0	-130	-152	-43	-109	-87	0
25	0	0	-26	-30	-9	-22	-17	0
26	0	0	-10	-12	-3	-9	-7	0
27	0	0	-10	-12	-3	-9	-7	0
28	-2	-2	-35	387	90	85	243	-3
29	0	0	-65	-76	-22	-54	-43	0
30	0	0	-65	-76	-22	-54	-43	0

Table 6. GBI Matrix for Relation #1
(Part 3 of 4)

Attr	17	18	19	20	21	22	23	24
1	140	125	0	0	0	-14	0	0
2	140	0	0	0	0	-144	0	0
3	0	0	0	0	0	-22	0	0
4	0	0	0	0	0	-22	0	0
5	-4	-13	-4	-22	74	107	109	-43
6	-13	86	-13	-65	-12	313	92	-130
7	-7	105	-7	-33	53	101	105	-65
8	-15	79	-15	-76	-34	367	87	-152
9	0	0	0	0	0	-12	0	0
10	0	0	0	0	0	-12	0	0
11	-13	-39	-13	-65	-12	96	92	-130
12	-15	79	-15	-76	-34	359	87	-152
13	-4	-13	-4	-22	-43	73	-9	-43
14	-11	-33	-11	-54	9	209	96	-109
15	-9	-26	-9	-43	31	362	101	-87
16	140	0	0	0	0	-23	0	0
17	0	0	0	0	0	-15	0	0
18	0	0	0	0	0	-46	0	0
19	0	0	0	0	0	-15	0	0
20	0	0	0	0	0	-76	0	0
21	0	0	0	0	0	-34	118	0
22	-15	-46	-15	-76	-34	0	87	-152
23	0	0	0	0	118	87	0	0
24	0	0	0	0	0	-152	0	0
25	0	0	0	0	0	-30	0	0
26	0	0	0	0	0	-12	0	0
27	0	0	0	0	0	-12	0	0
28	-2	119	-2	-11	-22	236	-4	-22
29	0	0	0	0	0	-76	0	0
30	0	0	0	0	0	-76	0	0

Table 6. GBI Matrix for Relation #1
(Part 4 of 4)

Attr	25	26	27	28	29	30
1	0	0	0	123	0	0
2	0	0	0	-14	0	0
3	0	0	0	4	0	0
4	0	0	0	4	0	0
5	-9	-3	-3	-7	-22	-22
6	-26	-10	-10	340	-65	-65
7	-13	-5	-5	114	-33	-33
8	-30	-12	-12	395	-76	-76
9	0	0	0	-2	0	0
10	0	0	0	-2	0	0
11	-26	-10	-10	-35	-65	-65
12	-30	-12	-12	387	-76	-76
13	-9	-3	-3	90	-22	-22
14	-22	-9	-9	85	-54	-54
15	-17	-7	-7	243	-43	-43
16	0	0	0	-3	0	0
17	0	0	0	-2	0	0
18	0	0	0	119	0	0
19	0	0	0	-2	0	0
20	0	0	0	-11	0	0
21	0	0	0	-22	0	0
22	-30	-12	-12	236	-76	-76
23	0	0	0	-4	0	0
24	0	0	0	-22	0	0
25	0	0	0	-4	0	0
26	0	0	0	-2	0	0
27	0	0	0	-2	0	0
28	-4	-2	-2	0	-11	-11
29	0	0	0	-11	0	0
30	0	0	0	-11	0	0

Table 7 GBI Matrix for Relation #6
(Part 1 of 2)

Attr	1	2	3	4	5	6	7	8
1	0	107	107	107	107	107	107	107
2	107	0	107	107	107	107	107	107
3	107	107	0	107	107	107	107	107
4	107	107	107	0	107	107	107	107
5	107	107	107	107	0	107	107	107
6	107	107	107	107	107	0	107	107
7	107	107	107	107	107	107	0	107
8	107	107	107	107	107	107	107	0
9	107	107	107	107	107	107	107	107
10	107	107	107	107	107	5722	107	107

11	107	107	107	107	107	107	107	107
12	107	107	107	107	107	107	107	107
13	107	107	107	107	107	86410	107	107
14	107	107	107	107	107	107	107	107
15	107	107	107	107	107	123355	107	107

Table 7 BI Matrix for Relation #6 (Part 2 of 2)

Attr	9	10	11	12	13	14	15
1	107	107	107	107	107	107	107
2	107	107	107	107	107	107	107
3	107	107	107	107	107	107	107
4	107	107	107	107	107	107	107
5	107	107	107	107	107	107	107
6	107	5722	107	107	86410	107	123355
7	107	107	107	107	107	107	107
8	107	107	107	107	107	107	107
9	0	107	107	107	107	107	107
10	107	0	107	107	107	107	107
11	107	107	0	107	107	107	107
12	107	107	107	0	107	107	107
13	107	107	107	107	0	107	86410
14	107	107	107	107	107	0	107
15	107	107	107	107	86410	107	0

Results

The clustering algorithm described in III.C is applied on the GBI matrices to generate the partitioning solutions. Table 8 gives the partitioning solution given by the proposed MRP procedure. The time taken by the MRP algorithm to solve this problem is 51.14 seconds on a Pentium PC, an average of 7.3 seconds per relation. The database access cost with MRP is 1632 milliseconds, while those with unfragmented (UN) and trivially fragmented (TR) are 226 milliseconds and 291 milliseconds, respectively. It represents cost savings over UN and TR of 28% and 44%, respectively.

The individual transaction costs for the entire database associated with the MRP, UN and TR designs are shown in Table 9. MRP solution keeps lower update costs (inserts/ deletes/ referential integrity maintenance) lower compared to TR solution and lower sequential access costs compared to UN solution. While UN has the highest sequential access cost, TR has the highest update costs and referential integrity checking costs.

Table 8. MRP Partitioning Solution

Relation Partitions
1.(8,12,6,22,14,11,13,18,1,23,5,7,28,15)(2,16,17,3,4,21)(9)(10)(19)(20)(24)(25)(26)(27)(29)(30)
2 (1,2,3)
3 (1,2,3,4,5)
4.(1,3-33, 35-42) (2,34)
5.(2,16,1,17,34,6,40)(3-15,18-33,35-39,41,42)
6 (1-15)
7 (1-29)

Table 9 Transaction Cost Analysis

Transaction costs	MRP	UN	TR
Sequental Retrievals	53	118	23
Random Retrievals	63	61	68
Inserts	21	3	81
Deletes	2	1	46
Referential Integrity Check	7	6	60
Joins	16	37	13
Total cost	**162 ms**	**226 ms**	**291 ms**

5. EVALUATION OF PARTITIONING PROCEDURE

Simulation Experiments

In order to validate the proposed MRP procedure, we performed simulation experiments under varying operating conditions. In the experiments, the size of the relational schema is set to 8 relations with 10 to 25 attributes each and tuples ranging from 10,000 to 20,000. The database/ transaction profiles used in the experiments are comparable to those of 'real-world' production database [34]. We ran 10 simulation experiments, with 5 experiments varying update proportion and another 5 varying proportion of join transactions. In the low (5%) update environment, our proposed MRP method showed cost savings of 53% over unfragmented design and 19% over trivial fragmentation design. In high update environment that is characteristic of volatile databases, the savings are 40% and 51% over unfragmented and trivially fragmented designs, respectively.

Comparison with Exhaustive Enumeration

The partitioning solutions by MRP method are compared with optimal partitioning design obtained by exhaustive enumeration, for small database problems. Table 10 shows the database access cost for MRP design and exhaustive enumeration. For these database problems, the database access cost by MRP is within 2% of the optimal access cost. The reason for obtaining a solution closer to optimal is that in smaller databases, search space is smaller and the heuristic algorithm more likely will arrive at optimal or near optimal solution. In larger databases, it may not be the case since the search space is larger and is difficult to verify the optimality. The time taken by MRP is of the order of $1/10^{th}$ second, while that of exhaustive enumeration is about 11000 seconds for a 4-relation schema with 4 attributes per relation. The execution time for 5 relations is 0.6 seconds with 10 attributes per relation, while it is 21.5 seconds with 34 attributes. As the number of transactions increase the execution time of MRP increased fairly linearly (0.5 sec with 500 transactions per hour and 0.6 sec for 1500 transactions per hour).

Table 10. Database Access Cost Comparison with Exhaustive Enumeration

	Attributes		
	3	4	5
Relation	Exhaust, MRP	Exhaust, MRP	Exhaust,MRP
3	44, 45	53, 54	----

4	49, 50	52, 54	----
5	49, 51	----	-----

Example from Previous Studies

Since there are no prior studies on multi-relational partitioning to compare, we construct a multi-relation schema with two relations that are drawn from previous studies. The first relation (R1) and the corresponding transactions are taken from the study of [28]; the second relation (R2) and the transactions are drawn from [8]. Table 11 gives description of the constructed database and Table 11 has the transaction set for the example.

Table 11. Database Description

Relation	Attr-ibutes	Records	Attribute Lengths
R1	10	10000	10 8 4 6 15 14 3 5 9 12
R2	20	10000	8 8 8 8 4 8 8 12 20 22 4 8 6 5 3 30 12 8 6 6

Referential integrity constraints

Relation	R1	R2
R1	0	1
R2	1	0

Table 12. Transaction Data

TranType	Rel	Freq	Records	Attributes
INSERT	R1	100	25	0
DELETE	R1	100	50	0
SEQUENCIAL	R1	100	0	3 4,6,10
SEQUENCIAL	R1	100	0	3 2,7,8
RANDOM	R1	100	25	7 1-3,5,7-9
RANDOM	R1	100	25	2 1,5
RANDOM	R1	100	25	3 3,4,9
RANDOM	R1	100	15	5 3,4,6,9,10
INSERT	R2	100	50	0
DELETE	R2	100	50	0
SEQUENCIAL	R2	100	0	6 3,7,10,11,17,18
SEQUENCIAL	R2	100	0	4 15,16,19,20
SEQUENCIAL	R2	100	0	3 1,5,8
RANDOM	R2	100	15	4 1,8,10,11
RANDOM	R2	100	15	5 3,7,10,11,17
RANDOM	R2	100	15	8 2,12-16,18,20
RANDOM	R2	100	10	5 2,5,11,14,19
RANDOM	R2	100	10	4 1,9,16,18
RANDOM	R2	100	10	8 1-6,9,12,13
RANDOM	R2	100	10	6 4,7,10,14,19,20
RANDOM	R2	100	10	6 8,11,15,16-18
RANDOM	R2	100	5	9 1-9
RANDOM	R2	100	5	6 15-20

The partitioning results of Song and Gorla [1] for relation R1 are four fragments: (1,5), (2,7,8), (3,4,9), and (6,10). The results of Cornell and Yu [8] for relation R2 are two fragments: (1, 3, 4, 5, 6, 7, 8, 10, 11, 17, 18) and (2, 9, 12, 13, 14, 15, 16, 19, 20). We modify the transaction data for R1 and R2 from the respective examples by making the first two transactions as insert / deletes and the next two transactions as sequential access transactions. This is done because, previous works do not need update transactions and sequential transactions are not inputs to their studies. The modified example has a transaction mix of 17% updates, 20% sequential access transactions, and 63% random access transactions. These are comparable to usual transaction loads in real-world relational databases [34]. Also we made access frequency of 100 for all transactions, while the previous research used 1 as the frequency. This should not affect the results and conclusions.

Our MRP design procedure resulted in three fragments for relation R1: (4, 6, 10), (1, 3, 5, 9), and (2, 7, 8), and four fragments for relation R2: (3, 7, 10, 11, 17, 18), (15, 16, 19, 20), (2, 4, 6, 9, 12, 13, 14), and (1, 5, 8). The procedure took 0.4 seconds for the 2-relation schema. With a cardinality of 10000 records for each relation, the access cost with the proposed design is 702 ms, compared to 838 ms for unpartitioned solution and 2247 ms for trivial partitioning, showing an improvement of 16% over unpartitioned solution, and 70% over trivial partitioning. Referential integrity maintenance cost amounted to 23% of total access cost for the proposed design.

With cardinality of 50000 for each relation, we obtain access time of 1324 ms for proposed design, 3441 ms for unpartitioned design, and 2885 ms for trivial partitioning. These imply the proposed solution has an improvement of 61% over unpartitioned and 54% over trivial partitioning. The high costs for the unpartitioned is mainly due to sequential access costs (772 ms for proposed design vs. 3253 ms for unpartitioned), which are inherent in large databases with high cardinality. The high costs for trivial partitioning is due to high update and referential integrity maintenance costs (444 ms for proposed design vs. 1874 ms for trivial partition). Two opposing factors resulted in fragmentation in our proposed design: sequential access transactions lead to more fragmentation and the update/referential integrity maintenance activities lead to less fragmentation. MRP design arrived at the 'best' design that balanced these factors.

7. CONCLUSIONS AND FUTUTE RESEARCH

This paper develops a methodology for vertical partitioning for a multi-relation database environment. Our proposed procedure builds on previous research of one-relation environment and extends substantially to multi-relation environment. The results demonstrate database access cost savings by the proposed method ranging from 36% to 62% over unfragmented design, and from 19% to 54% over fully fragmented design. Referential integrity enforcement cost is found to be a major cost overhead in medium to high update environments. The access costs obtained with the proposed approach are comparable to the optimal partitioning solutions obtained by exhaustive enumeration for small problems. The complexity of the proposed clustering algorithm is $O(m*n^2)$, where m is the number of relations and n is the number of attributes per relation. The proposed procedure produced partitioning solutions within a few seconds for most database problems. The proposed methodology can be applied to partitioning problems in distributed and object-oriented databases. Since the update transactions tend to cause database integrity problems, the DBMS (Distributed DBMS or Object-Oriented DBMS) generates additional transactions to ensure data consistency. As discussed in the paper, these additional transactions have implications for effective attribute partitioning.

The research can be extended in several directions. First, we assumed in this research that the transactions that

operate on the relational schema are static. Periodically, say once a year, the vertical partitioning can be derived using the change in transaction mix. Present research can be extended to derive the vertical partitioning dynamically, based on the incremental changes in the transaction mix. Thus incremental changes to the vertical partitions can be derived, which will make the physical design optimal all the time. Adaptive designs are shown to have better performance than static designs in database design [33]. Second, the research can be extended to include concurrency control into the model and its effect on the response time model. Including concurrency control mechanism, similar to enforcement of referential integrity constraints, imposes additional database transactions on the database records, and affects both 'optimal' fragment design and access time performance with partitioned databases. By considering the concurrency control issues, the proposed research can be made even more comprehensive. Third, in this research we assumed the access paths as given. This research can be extended to include the access path determination and to solve the combined problem of attribute partitioning and access path selection in a multi-relation environment. The methods used by [8,28] can be starting points, since they specify access paths to one or two fragments. Thus the multi-relation partitioning case can be extended to include access paths for all the partitions, along with query processing schedule using those partitions and access paths. Fourth, attribute affinity can be computed for attributes of different relations, rather than in those for attributes within each relation. The affinity between attributes could be high due to enforcement of referential integrity constraints and due to join transactions. The design of storage structures using this approach is another area of potential extension to our research. Fifth, presently we are working on mixed fragmentation procedure, where in vertical partitioning and tuple clustering are determined simultaneously. Similar to attribute partitioning, record clustering involves assignment of records to blocks. Since these two problems are interdependent, an iterative procedure is being developed that will solve the combined problem.

Appendix A
Access-Cost Function

$$COST = t_{br} \cdot \sum_k F^{sk}_{T(r)} \sum_r c(r) \sum_{f(r)} \sum_{i(r)} (L_{i(r)} \cdot z^{i(r)}_{f(r)}) \cdot x^k_{f(r)} / B$$

(sequential retrieval cost)

$$+ t_{ar} \cdot \sum_k F^{rk}_{T(r)} \cdot \sum_r \sum_{f(r)} ACC^{rk}_{f(r)} \cdot x^k_{f(r)}$$

(random retrieval cost)

$$+ \sum_k F^m_{T(r)} \cdot \sum_r \sum_{f(r)} (u^k_r \cdot t_{br} \cdot c(r) \cdot (\sum_{i(r)} L_{i(r)} \cdot z^k_{f(r)})/B + v^k_r \cdot t_{ar} \cdot ACC^{rk}_{f(r)}) \cdot x^k_{f(r)}$$

(join cost)

$$+ (t_{ar} + t_{rw}) \cdot \sum_k \sum_r F^{ik}_{T(r)} \cdot AVR^{ik}_{T(r)} \cdot NS(r) \cdot p^k_r$$

(insertion cost)

$$+ (t_{ar} + t_{rw}) \cdot \sum_k \sum_r F^{dk}_{T(r)} \cdot AVR^{dk}_{T(r)} \cdot NS(r) \cdot p^k_r$$

(deletion cost)

$$+ (t_{ar} + t_{rw}) \cdot \sum_k \sum_r F^{dk}_{T(r)} \cdot AVR^{dk}_{T(r)} \cdot \sum_{m\,(m<>r)} NS(m) \cdot p^k_r \cdot y_{rm}$$

(referential integrity cost for deletions)

$$+ t_{ar} \cdot \sum_k \sum_r F^{ik}_{T(r)} \cdot AVR^{ik}_{T(r)} \cdot \sum_{m\,(m<>r)} p^k_r \cdot y_{mr}$$

(referential integrity cost for insertions)

where

$NS(r)$ = No. of fragments in relation r.

$z^{i(r)}_{f(r)}$ = 1 if attribute i of relation r is assigned to fragment f of relation r, 0 otherwise.

$x^k_{i(r)}$ = 1 if k^{th} transaction uses i^{th} attribute of relation r, 0 otherwise.

$$ACC^{rk}_{f(r)} = (1 - (1 - B/c(r).\Sigma_{i(r)} L_{i(r)}.z^{i(r)}_{f(r)})^{AVR^{rk}_{T(r)}}).c(r).\Sigma_{i(r)} L_{i(r)}/B$$

p^k_r =1 if k^{th} insert/delete transaction operates on relation r, 0 otherwise.

u^k_r =1 if k^{th} Join transaction needs a sequential access on relation r, 0 otherwise.

(v^k_r same as above for random access)

y_{rm} =1 if m is a referenced relation to the primary relation r, 0 otherwise.

$AVR^{qk}_{T(r)}$ = Total no. of records accessed in relation r by k^{th} transaction of type q.

8. REFERENCES

1. Ailamaki, A; Dewitt, D.J.; Hill, M.D. and Skounakis, M.,"Weaving Relations for Cache Performance," *Proceedings of the 27th VLDB Conference*, 2001
2. Baiao, F; Mattoso, M and Zaverucha, G., "A Distribution Design Methodology for Object DBMS," Journal of Distributed and Parallel Databases, 16 (6), 2004, 45-90
3. Cardenas, A.F., "Analysis and performance of inverted database structures", Communications of the ACM 18, 5, (May 1975), 253-263.
4. Ceri, S., Navathe, S., and Wiederhold, G., "Distribution Design of Logical Database Schemas", IEEE Trans. Soft. Eng. SE-9, 4, (July 1983)
5. Cheng, C-H; Lee, W-K; Wong, K-F, "A Genetic Algorithm-Based Clustering Approach for Database Partitioning," *IEEE Transactions on Systems, Man, and Cybernetics*, 32(3), 2002, 215-230.
6. Chu, P., "A Transaction-Oriented Approach to Attribute Partitioning," Information Systems, vol. 17, no. 4, 1992, pp 329-342.
7. Chu, W. W. and Ieong, I.T., "A Transaction-Based Approach to Vertical Partitioning for Relational Database Systems", IEEE Transactions on Software Engineering, 19-9, August 1993.
8. Cornell, D.W. and Yu, P.S., "An Effective Approach to Vertical Partitioning for Physical Design of Relational Databases", IEEE Transactions on Software Engineering, 16-2, (Feb 1990)
9. Date, C.J., An Introduction to Database Systems. Addision-Wesley Publishing Company, Sixth Edition, (1995).
10. Ezeife, C.I., "Selecting and materializing horizontally partitioned warehouse views," Data and Knowledge Engineering, 36, 2001, pp 185-210
11. Elmasri, R. and Navathe S. B., Fundamentals of Database Systems, The Benjamin/ Cummings Publishing Company, Inc., Second Edition, 1994.
12. Fung, C-w; Karlapalem, K. and Li, Q., "An Evaluation of Vertical Class Partitioning for Query Processing in Object-Oriented Databases," *IEEE*

Transactions on Knowledge and Data Engineering, 14(5), 2002, 1095-1118.
13. Furtado, C; Lima, A.A.B.; Pacitti, E; Valduriez, P. and Mattoso, M., "Physical and virtual partitioning in OLAP database cluster," 17th International Symposium on Computer Architecture and High Performance Computing, 2005, pp 143-150
14. Gorla, N., "An Object-oriented database design for improved performance," Data & Knowledge Engineering, 2001.
15. Gorla, N. and Liu, C., "FHIN: an efficient storage structure and access methods for object-oriented databases," Information and Software Technology, vol. 41, 1999, pp. 673-688.
16. Hammer, M., and Niamir, B. "A Heuristic Approach to Attribute Partitioning", ACM SIGMOD International Conference on Management of Data (1979).
17. Hoffer, J.A. and Severance, D.G. "The Use of Cluster Analysis In Physical Data Base Design", International Conference On Very large Databases (1975).
18. Labio, W.J., Quass, D., and Adelberg, B., "Physical Database Design for Data Warehouses, IEEE Conference on Data Engineering, 1997, pp 277-288.
19. Lim, S-J and Ng, Y-K, "Vertical Fragmentation and Allocation in Distributed Deductive Database Systems," Information Systems, vol. 22, No. 1, 1997, pp 1-24.
20. Mannino, M.V., Database Design, Application Development, and Administration. McGraw-Hill, Third Edition, 2007
21. March, S.T. "Techniques for Structuring Database Records", ACM Computing Surveys 15, 1, 1983.
22. March, S.T. and Rho, S., "Allocating Data and Operations to Nodes in Distributed Database Design," IEEE Trans on Knowledge and Data Engineering, vol. 7, no. 2., 1995, pp 305-317.
23. Navathe, S., Ceri, S., Wiederhold, G., and Dou, J. "Vertical Partitioning Algorithms for Database Design", ACM Trans. Database Syst. 9, 4 (Dec. 1984). 680-710.
24. Navathe, S and Ra, M. "Vertical Partitioning for Database Design: A graphical algorithm", Proceedings of ACM SIGMOD, 1989.
25. Ng, V; Gorla, N.; Law, D.M. and Chan, C.K., "Applying Genetic Algorithms in Database Partitioning," Proceedings of the 2003 ACM Symposium on Applied Computing (SAC) 2003, pp 544-549.
26. Ozsu, M. and Valduriez, P., Principles of Distributed Database Systems, Prentice Hall, 1996.
27. Ramamurthy, R; Dewitt, D.J. and Su, Q., "A Case for Fractured Mirrors," *Proceedings of the 28th VLDB Conference*, 2002
28. Song, S.K. and Gorla, N., "A genetic Algorithm for Vertical Fragmentation and Access Path Selection," The Computer Journal, vol. 45, no. 1, 2000, pp 81-93.
29. Stonebraker, M., Aoki,P.M., Litwin, W. and Olson, M., "Mariposa: A Architecture for Distributed Data," 10th International Conference on Data Engineering, 1994, pp 54-65.
30. Tamhankar, A.J. and Ram, S., "Database Fragmentation and Allocation: An Integrated Methodology and Case Study," IEEE Trans on Systems, Man, and Cybernetics – Part A, May 1998, pp 288-305.
31. Wiederhold, G., File Organization for Database Design. McGraw-Hill Company, 1987.
32. Yao, S.B. "Approximating Block Accesses in Database Organizations", CACM 20, 4,1977.
33. Wolfson, Ouri; Jajodia, Sushil; Huang, Yixiu, "An Adaptive Data Replication Algorithm," ACM Transactions on Database Systems, vol. 22, no. 2, June 1997.
34. Yu, P.S., Chen, M-S, Heiss, H-U, and Lee, Sukho, "On Workload Characterization of Relational Database Environments," IEEE Trans. Software Engineering, vol.18, no. 4, April 1992, pp 347-355.

Received: May 2007. Accepted: Jun. 2007.