An Optimized Scheme for Vertical Partitioning of a Distributed Database

Eltayeb Salih Abuelyaman
CCIS, Prince Sultan University, Riyadh 11586, Saudi Arabia

Summary

This paper proposes a scheme for vertical partitioning of a database at the design cycle. When a partition is formed, attributes are divided among various systems or even throughout different geographical locations. This may result in situations where a query may include attributes that are located at different sites. The scheme determines the hit ratio of a partition. As long as it falls below a predetermined threshold, the partition is altered. Although no proof is provided, experimental data showed that moving an attribute that is loosely coupled to a different subset within a partition improves hit ratio. A simulator was built to test the proposed algorithm. Results of various simulation runs are consistent with the hypothesis. That is, the proposed algorithm enables a reliable distribution of newly designed database tables across multiple storage devices based on a predetermined hit ratio. The scheme is independent of frequencies of queries thus, can be used as a stepping stone for its counterpart, the dynamic partitioning technique.

Keywords: *database, partition, frequency, query, reflexive, symmetry, transitivity, hit ratio.*

1. Introduction

Distributed and parallel processing is an efficient way of improving performance of Data-Base Management Systems (DBMSs) and applications that manipulate large volumes of data. Such improvement comes from limiting queries only to data that are relevant to their respective transactions. This is one of the main design goals of distributed databases according to [2].

The primary concern of DBMS design is the fragmentation and allocation of the underlying database. The distribution of data across various sites of computer networks involves making proper fragmentation and placement decisions. The first phase in the process of distributing a database is fragmentation which clusters information into fragments. This process is followed by the allocation phase which distributes, and if necessary, replicates the generated fragments among the nodes of a computer network. The use of data fragmentation to improve performance is not new and commonly appears in file design and optimization literature [3].

Partitioning based on attributes has been studied earlier in [3], [4], [6], [8]. Stocker and Dearnley discussed implementation of a self-reorganizing DBMS that carries out attribute clustering [9]. They showed that it is beneficial to cluster attributes of a DBMS where storage cost is low compared to the cost of accessing subfiles. Such is the case because increases in storage costs will be offset by savings in access cost. Hoffer developed a non-linear, zero-one program which minimizes a linear combination of the costs of: storing, retrieving and updating, with capacity constraints for each file [6]. Navathe et al used a two-step approach for vertical partitioning. In the first step, they used the given input parameters in the form of an Attribute Usage Matrix (AUM) to construct an Attribute Affinity Matrix (AAM) for clustering [8]. After clustering, an empirical objective function is used to perform binary partitioning iteratively. In the second step, estimated storage cost factors are considered for further refinement of the partitioning process. Further details about AUM and AAM matrices will be provided in the next paragraph.

Cornell and Yu extended Navathe et al approach to decrease the number of disk accesses for optimal binary partitioning [5]. Their extension involved specific physical factors such as: the number of attributes, their length and selectivity, the cardinality of the relation and so on. Navathe and Ra developed a new algorithm that follows graph

theory partitioning techniques [7]. Their algorithm starts from the AAM matrix, which is transformed into a graph called the Affinity Graph (AG). An edge in AG is labeled with a weight that represents the affinity between its vertices, where: vertices represent attributes; affinity between vertices represents the number of queries in which the attributes occurred simultaneously. For the interest of clarity of presentation we will define what an AAM matrix is. For more details, interested readers are referred to reference [8]. Basically, an *n x n* AAM matrix is one whose AAM(*i, j*) entry represents the number of queries that simultaneously access the attributes represented by *i* and *j*. Based on the AAM, an iterative binary partitioning method has been proposed in [5 and 8]. The authors first clustered the attributes and then applied empirical objective functions and/or mathematical cost functions to perform fragmentation.

The partitioning algorithms suggested in the literature suffer from various limitations that will complicate the task of a Data Base Designer (DBD). These limitations are:

1. A DBD has to have sufficient empirical data on Frequencies Of Queries (FOQ).
2. FOQ is a function of several variables that include time, users, and future needs of an organization.
3. Attributes are partitioned based on FOQ.

The first limitation makes the partitioning inapplicable to newly designed database schemas. The second applies to periodical queries the likes of those for student records and taxes databases. Furthermore, changes in organizational structures or business requirements may call for additional attributes. The third limitation stalks from the natural dynamicity of FOQ.

The common denominator in all three limitations is FOQ. Therefore, the author herein classifies all partitions that are based on FOQ as dynamic. On the other hand, the only way for a partition to be independent of FOQ is when it is based on a database schema. In this case it will be logical to classify it as static. The proposed partition is static and will be called **StatPart.**

The rest of this paper is organized as follows: In the next section we will introduce the StatPart. In section 3 we will discuss simulation of StatPart. A comparison of performance of StatPart with a benchmark is given in Section 4. The conclusion is presented in sections 5.

2. StatPart

In general, designers very frequently delay important steps to the end of design cycles. The design of efficient database systems is not an exception because database partitioning is based on FOQ. That is, data must be collected from a large number of queries before partitioning. To circumvent such a constraint, dependency on FOQ must be eliminated. One way to do so is to perform database partitioning at the design phase and immediately after completion of the schema. Conveniently, partitioning can be decided even before database tables are populated. For such a partition, which we classified as static, the DBD must:

1. Gain sufficient knowledge on the business requirements of an organization.
2. Gather necessary and sufficient information about intended usage of the database to determine the set of queries that would be of immediate use. Henceforth, this set will be called the Set of Kickoff Queries (SKQ). This step requires thorough understanding of the business requirement of an organization.
3. Gather information about future plans of an organization to determine additional queries that may be needed in the future. This set will be referred to as the Set of Future Queries (SFQ).

For illustration of the proposed static partitioning, the following definition will be necessary.

Definition 1:

a) N_a : the total number of attributes.
b) N_k : the number of queries in the set SKQ.
c) N_f : the number of queries in the set SFQ.
d) S_Q: the union of the set SKQ with SFQ.
e) $S_A = \{A_1, A_2 \ldots A_{Na}\}$**:** the overall set of attributes.
f) $S_Q = \{Q_1 , Q_2, \ldots , Q_{Nq}\}$: the overall set of queries.

In the next section, we will discuss a suggested simulator for the static partitioning.

3. Simulation of StatPart

Our proposed simulator will enable a DBD to partition a database at its infancy, that is, at its

schema level. The output of the simulator may range from **0** to **100** percent where **0** percent implies that the schema cannot be partitioned and **100** percent means that every attribute is placed in a partition by itself. Both percentages are undesirable and the latter is unacceptable. Along with the schema, a complete set of queries and parameters defining the database must be fed into the simulator. These parameters will be discussed hereafter as necessary.

The simulator has the following three modules. Each of the modules is discussed separately.

a) **reflexivity**
b) **symmetry**
c) **transitivity**

3.1 The *reflexivity* module

The module prompts a user to enter values for each of the first three parameters in Definition 1. The module then prompts the user to enter a percentage **C** that controls the number of attributes appearing in each query. If the designer enters the value **30** for example, then the module will generate a value of **1** with probability of **0.3** and a value of **0** with probability of **0.7**. That is, if the total number of attributes is **10** then on the average a query would include three attributes. The ***reflexivity*** module will then generate a matrix that relates attributes to queries. For **C = 30** and **(N_a , N_k , N_f) = (8,5,3)**, we found the output challenging enough to use for the interest of the discussion. The output is shown on Table 1 below. Henceforth, such output will be called the ***Reflexivity Matrix*** (RM).

In an **RM** matrix, the total number of 1's on a column gives the degree of ***reflexivity*** of the column header's attribute. For example, in table 1 the reflexivity of attribute **B** is equal to **3**.

Query \ Attribute	A	B	C	D	E	F	G	H
a	**0**	**1**	**0**	**0**	**1**	**0**	**1**	**1**
b	**0**	**0**	**1**	**1**	**0**	**1**	**0**	**0**
c	**1**	**0**	**1**	**1**	**0**	**0**	**0**	**0**
d	**0**	**0**	**0**	**1**	**1**	**1**	**1**	**1**
e	**0**	**1**	**1**	**0**	**1**	**1**	**1**	**0**
f	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
g	**0**	**1**	**1**	**0**	**0**	**1**	**1**	**1**
h	**0**	**0**	**1**	**1**	**1**	**1**	**0**	**0**

Table 1. A randomly generated Reflexivity Matrix

Once again, a **0** entry on the table indicates that the row header query does not involve the column header attribute. One can see that **RM [d, F]** is equal to **1**. The table provides the relationship between queries and attributes. However, it doesn't directly provide desired relationships among attributes. The **RM** matrix, however, will be used as input to the second module, the ***symmetry*** module, which will produce the desired relationship.

3.2 The *symmetry* module

The following equations were used to compute the ***Symmetry Matrix*** (SM) on table 2 which defines the desired relationships among attributes.

$$SM[j, j] = \sum_{i=1}^{Na} RM[i, j] \quad \text{for } j = 1 \text{ to } N_a \qquad (1)$$

$$SM[i, j] = \sum_{k=1}^{Na} RM(k, i) * RM(k, j)$$

For $i = 1$ to N_a (For $j = 1$ to N_a) $i \neq j$ (2)

Attribute \ Attribute	A	B	C	D	E	F	G	H
A	**1**	**0**	**1**	**1**	**0**	**0**	**0**	**0**
B	**0**	**3**	**2**	**0**	**2**	**2**	**3**	**2**
C	**1**	**2**	**5**	**3**	**2**	**4**	**2**	**1**
D	**1**	**0**	**3**	**4**	**2**	**3**	**1**	**1**
E	**0**	**2**	**2**	**2**	**4**	**3**	**3**	**2**
F	**0**	**2**	**4**	**3**	**3**	**5**	**3**	**2**
G	**0**	**3**	**2**	**1**	**3**	**3**	**4**	**3**
H	**0**	**2**	**1**	**1**	**2**	**2**	**3**	**3**

Table 2. Symmetry Matrix generated from the Reflexivity Matrix and equations 1 and 2

Equation 1 adds up column entries for each attribute **j** in table 1 to determine its ***reflexivity***. The diagonal entries on an **SM matrix** give the ***reflexivity*** degrees of attributes. Equation 2 finds the intersection between each pair of attributes **i** and **j**. For example, in table 1, if we performed entry by entry multiplication of attributes **i** = E = $(1,0,0,1,1,0,0,1)^T$ and **j** = F = $(0,1,0,1,1,0,1,1)^T$ the result would be **i*j** = $(0,0,0,1,1,0,0,1)^T$. Entries in the result add up to 3 which is the value stored in both **SM[E,F]** and **SM[F,E]** of table 2.

If the matrix is transformed into a graph, then without loss of generality we assume attribute **K** to be represented by vertex **V**. The weight of the edge

connecting **V** with a vertex **W** is given by the ***symmetry*** value of attribute **K** with the attribute represented by **W**. On the other hand, ***reflexivity*** for an attribute indicates that an edge starts from, and ends at the same vertex.

It is practical to assume that each attribute is included in at least one query. Consequently, each must have a ***reflexivity*** degree of at least one. From table 1 in the previous section, the ***reflexivity*** of attribute **B** was found equal to **3**. This ***reflectivity*** degree is stored in **SM[2,2]** of table 2. Every diagonal element gives the ***reflectivity*** degree of its column header's attribute. A non-diagonal entry of **SM** gives the ***symmetry*** between the row and column headers and is equal to the number of queries that include both. The **SM** matrix can be thought of as a data structure for a graph. Consequently, for the corresponding graph, the ***symmetry*** is modeled by an edge connecting the two vertices (attributes). The **SM** matrix is itself symmetrical around its diagonal. At this point, we are ready to discuss the ***transitivity*** module which acts upon the output of the ***symmetry*** module.

3.3 The *transitivity* module

Before proceeding with further discussions, we will summarize the above concepts in the following definitions.

Definition 2

a) ***Reflexivity***: Reflexivity of an attribute **Z** represents the number of queries that reference **Z**. In an **RM** matrix, the number of 1's on the column of attribute **Z** represents the degree of ***reflexivity*** of **Z**. On the corresponding graph, Reflexivity is represented by an edge that loops back to the same vertex and the Degree of Reflexivity (**DR**) is the label on the edge.

b) ***Symmetry***: Two attributes are called symmetric if there is at least one query that includes both. On an SM's corresponding graph, ***symmetry*** between any two vertices **U** and **W** is represented by an edge connecting **U** to **W**. The Degree of Symmetry (**DS**) between **U** and **W** is represented by the label on their edge, and corresponds to the number of queries that include both.

Prior to discussing the algorithm, we will state that for any attribute (or vertex) **V**, its **DR** will always be greater than or equal to each of the **DS** values with neighboring vertices (attributes). This is true because from definition 1 above, **DR** gives the number of queries referencing **V** while for any other attribute **W**; **DS** gives the number of queries referencing both **V** and **W**. The ***transitivity*** module receives the **SM** matrix as input and produces the required partition as output. However, before discussing the ***transitivity*** module, we will first plan a strategy that sets the mechanism for the algorithm and then suggest tactics to optimize its output.

3.4 Strategy and tactics

In general, the success of an algorithm depends on the strategy that sets criteria for choosing the best start point and the smartest move thereafter. Going back to the graph theory terminology, one must first choose the best vertex to start with, and then pick the most appropriate edge to traverse in search for an optimal partition. Equally, our strategy will focus on selecting the attribute to start with, which is a function of the **DR**, and then picking the best neighbor to reach, which is a function of **DS**.

To that end, there are four different possibilities: **{(+DR,+DS);(+DR,–DS);(–DR,+DS);(–DR,–DS)}**, where **+DR (+DS)** represents the maximum degree of ***reflexivity*** (***symmetry***) of the remaining vertices (attributes). Similarly, **–DR(–DS)** represents the minimum degree of ***reflexivity*** (***symmetry***). Based on our empirical data, we determined that the most appropriate combination is the **(–DR,+DS**) pair.

The algorithm starts with a vertex **V** that satisfies (–DR). We will call **V** the ***current*** vertex. The algorithm then finds a vertex with **+DS** among **V**'s neighbors. Once such a neighbor is found, both **V** and its neighbor are placed in a subset. The neighbor would then become the ***current*** vertex. The process would continue to search neighbors of the most recent ***current*** vertex in a similar manner until a cycle is formed or no more vertices are left. The algorithm will continue to produce new subsets in the same manner and ends when all vertices are handled. The resulting subsets are disjoint and together represent the partition.

The next step is to compute the partition's **hit ratio**. A **hit ratio** of one hundred percent occurs when all attribute are in a single set. The only time this is true is when the schema cannot be partitioned. A DBD is responsible for setting up a partition's **hit ratio** threshold.

The tactic of the algorithm is to look for the most loosely coupled attribute in the partition and move it to a different subset. The new **hit ratio** is then

computed and checked against the threshold. The process is continued until an acceptable **hit ratio** is achieved. A simplified version of the partitioning algorithm is shown on figure 1 below.

*a) From the **SM** matrix, select the attribute **V** that corresponds to **–DR**. Add **V** to the temporary subset **S**, Remove **V** from the set of attributes **As***

*b) From the attributes adjacent to the current **V** select the attribute that corresponds to **+DS**, say attribute **W**. Remove **W** from **As** and add it to **S***

*c) repeat step (b) but with **W** as the new **V** to find a new **W**. The process continues until the set **As** is empty or a cycle is formed*

*d) If a cycle has been formed then copy the attributes in **S** as a subset in the partition set **P***

*e) repeat from step (b) if **As** is not empty and a new **V** with **–DR** exists*

*f) The subsets in **P** form a partition*

Figure 1. A simplified version of the partitioning algorithm

3.5 Illustration of the tactics

To illustrate the process, we used the **SM** on table 2 to produce the partition **P**.
Initially **As** is equal to **{(A, B, C, D, E, F, G, H)}.** The following are step by step execution of the algorithm.

(a) **S = {A} and As = {(B, C, D, E, F, G, H)}**
(b) **S = {(A,C)} and As = {(B, D, E, F, G, H)}**
(c) **S = {(A,C,F)} and As = {(B, D, E, G, H)}**
(d) **S = {(A,C,F,D)} and As = {(B, E, G, H)}**

P = { (A, C, D, F) ; (B, E, G, H) }

The **hit ratio** for the partition can be computed from the following table which shows the number of times an attribute is associated with its subset mates (**hits**) verses the number of times it is associated with others that are not in its subset (**misses**).

	A	B	C	D	E	F	G	H	Total
hit	2	7	8	7	7	7	9	7	54
miss	0	4	7	4	7	10	6	4	42

Table 3. Attribute associate for P

To demonstrate how the entries of table 3 are computed, let us examine the second row in table 2, which gives association of attribute **B**. The valid nonzero entries are **(2, 2, 2, 3, 2)** falling under column headers **(C, E, F, G, H)** respectively. These entries give the nonzero **DS** values of neighbors of attribute **B**. The partition that we obtained in the first round which was given above, is repeated for convenience of the reader as follows:
P = { (A, C, D, F) ; (B, E, G, H) }
Now the **hit** for attribute **B** is found by adding its **DS** values with members of its subset **[E, G, H]**. In this case, the **hit** value is **(2+2+3)** or **7**. Since **B** is also associated with **[C, F]** who are not members of its subset. The miss value is **(2+2)** or **4**. Table 3 shows under **B**'s column these values.

A closer look at table 3 reveals that **F** is the attribute with the worst **hit** to **miss** ratio. That is, **F** is loosely coupled with its subset mates. If the threshold is set at **60%** then the partition **P** must be altered. Moving **F** to the second subset in the partition would reverse the worst attribute **hit ratio** on the table from **(7:10)** to **(10:7)**. Now **F** is tightly coupled with its new subset mates. However, the overall impact of this change is still unclear. The new **hit ratio** for the altered partition **P'** must therefore be computed. Table 4 shows these values where **P'** is:

P' = { (A, C, D) ; (B, E, F, G, H) }.

	A	B	C	D	E	F	G	H	Total
hit	2	9	4	4	10	10	12	9	60
miss	0	2	11	7	4	7	3	2	36

Table 4. Attribute associate for P'

Table 4 reflects an improvement of the partition's **hit ratio** from **56.3%** to **62.5%**. Each attribute's **hit**(**miss)** value in table 3 differ from its corresponding value in table 4 by the **DS** the attribute shares with **F**. The strategy can now be summarized as follows in figure 2.

1. Produce a partition from an **SM** using the criterion **(–DR, +DS)** and figure 1.
2. Compute the partition **hit ratio** (**PHR)**
3. If **PHR** is less than the predefined threshold then
   a) Find the attribute with the minimum **hit** to **miss** ratio and move it to a different subset using the attribute association table in the process
   b) Repeat from step (2)
4. End partitioning

Figure 2. Partitioning Strategy

4. StatPart VS a benchmark

We will use the Attribute Usage Matrix (**AUM**) in reference [8] as a benchmark and compare the results of their example with the output of our simulator. Their **AUM** matrix is shown on table 5 below.

Queries \ Attributes	A	B	C	D	E	F	G	H	I	J
a	1	0	0	0	1	0	1	0	0	0
b	0	1	1	0	0	0	0	1	1	0
c	0	0	0	1	0	1	0	0	0	1
d	0	1	0	0	0	0	1	1	0	0
e	1	1	1	0	1	0	1	1	1	0
f	1	0	0	0	1	0	0	0	0	0
g	0	0	1	0	0	0	0	0	1	0
h	0	0	1	1	0	1	0	0	1	1

Table 5. The AUM matrix from reference[8]

One can see that the **AUM** matrix is quite similar to the **RM** matrix in table 1. First, we will use our ***symmetry*** module to produce an **SM** matrix which would serve as input to the ***transitive*** module. The latter will produce a partition that can be compared with the partition in the reference [8].

Based on the **AUM** matrix as input, the output of the ***symmetry*** module is shown in table 6 below. The table will be referred to as the ***Symmetry*** Matrix for **AUM**. Entries of **SM** are computed using equations 1 and 2.

Attributes \ Attributes	A	B	C	D	E	F	G	H	I	J
A	3	1	1	0	3	0	2	1	1	0
B	1	3	2	0	1	0	2	3	2	0
C	1	2	4	1	1	1	1	2	4	1
D	0	0	1	2	0	2	0	0	1	2
E	3	1	1	0	3	0	2	1	1	0
F	0	0	1	2	0	2	0	0	1	2
G	2	2	1	0	2	0	3	2	1	0
H	1	3	2	0	1	0	2	3	2	0
I	1	2	4	1	1	1	1	2	4	1
J	0	0	1	2	0	2	0	0	1	2

Table 6. The SM matrix for the AUM matrix

Now that we have the **SM** matrix for **AUM**, we will use it as input to the ***transitivity*** module to determine the partition. As in the previous section, we used the algorithm in figure 1 to produce a partition. We call the resulting partition **Q** to distinguish it from the partition in the previous example.

Q = { (A, B, C, E, G, H, I) ; (D, F, J) }

The next step is to compute the partition's **hit ratio**. In a similar manner as before, table 7 is produced as the attribute association table for **Q**.

	A	**B**	**C**	**D**	**E**	**F**	**G**	**H**	**I**	**J**	**Total**
hit	12	14	15	2	12	2	13	14	15	2	101
miss	0	0	3	6	0	6	0	0	3	4	22

Table 7. Attribute associate for

The partition **hit ratio** is **82%,** which is above the threshold of **60%.** The partition for the **AUM** matrix from reference [8] is shown in figure 3 below. Replacing the numbers from **1** to **10** in the figure with the letters **A** through **J** we can see that the two partitions are identical. The figure suggest that **(4, 6, 10)** should be in a subset while the rest in another. Equally, our partition put **(D, F, G)** in a different subset than that of the rest.

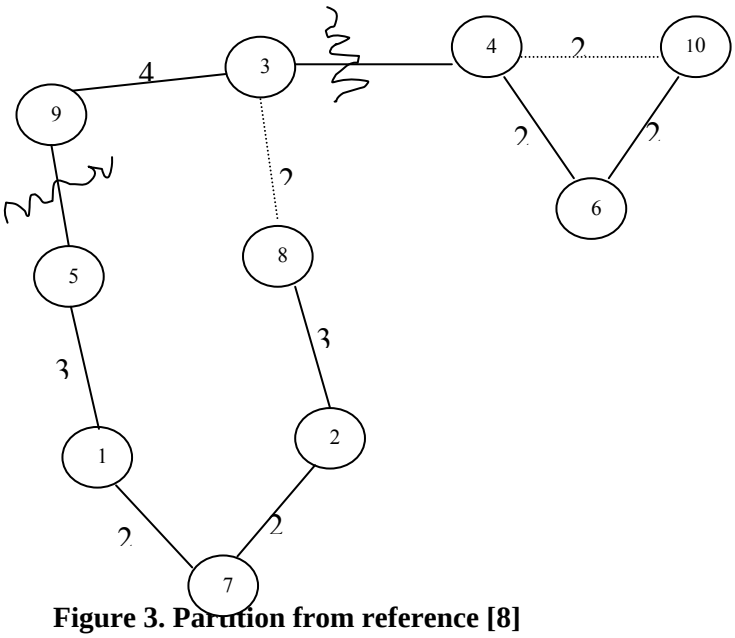

Figure 3. Partition from reference [8]

An important point to mention is that the algorithm presented in this paper is a simplified version of the one used for simulation. Few steps are not included in this document because the reader can easily figure them out using his/her imagination and creativity.

5. Conclusions

This paper proposed a vertical partitioning algorithm for improving the performance of database systems. The algorithm uses the number of occurrences of an attribute in a set of queries rather than the FOQ accessing these attributes. This enables the fragmentation of a database schema even before its tables are populated. Thus, a database designer will be in a position to perform partitioning and consequent distribution of fragments before the database enters operation. A simulator for the algorithm has been written. Results of simulations were consistent with those obtained using frequency based partitioning algorithms. The significant advantage of the proposed algorithm is that a database designer doesn't have to wait for empirical data on query frequencies before partitioning a database.

References

[1] H. Abdalla and M. AlFares, "Vertical Partitioning for Database Design: A Grouping Algorithm", to appear in SEDE 2007.

[2] M. Özsu and P. Valduriez, Principles of Distributed Database Systems, 2nd edition (1 st edition 1991), New Jersey, Prentice-Hall, 1999.

[3] M. Babad. A record and file partitioning model. Commun. ACM 20, 1(Jan 1977).

[4] F. Baião "A Methodology and Algorithms for the Design of Distributed Databases using Theory Revision"
D.Sc. Thesis, COPPE/UFRJ, Dec 2001. (http://www.cos.ufrj.br/~baiao/thesis/baiaoDSc.pdf).

[5] D. Cornell, and P. Yu. A Vertical Partitioning Algorithm for Relational Databases. Proc. Third International
Conference on Data Eng. , Feb. 1987.

[6] J. Hoffer. An integer programming formulation of computer database design problems. Inf. Sci., 11(July 1976), 29-48.

[7] S. Navathe, and M. Ra. Vertical Partitioning for Database Design: A Graphical Algorithm. ACM SIGMOD, Portland, June 1989.

[8] S. Navathe, S. Ceri, G. Weiderhold, and J. Dou. Vertical Partitioning Algorithms for Database Design ACM Transactions on Database Systems, Vol. 9, No. 4, 1984.

[9] M. Stocker and A. Dearnley. Self-organizing Data Management Systems Computer Journal. 16, 2(May 1973).

[10] H. Abdulla , E. Abuelyaman and F. Marir "A Static Attribute-Based Partitioning Algorithm(SAPA) for Vertical Fragmentation Problem in DDBs" Proceedings of the 2007 International Conference on Parallel and Distributed Processing Techniques and Applications, Volume II, pp 1017-1022, Las Vegas, 2007

Eltayeb Salih Abuelyaman received a PhD degree in Computer Engineering from the University of Arizona in 1988. He served as faculty member at various universities in the US for 18 years before moving to Prince Sultan University in Saudi Arabia where he served as a Faculty Member, a Director of the Information Technology and Computing Services and currently serves as the Dean of the College of Computer and Information Sciences. His current research Interest is in the areas of Computer Networks and Information Security and Database.