An Overview of Vertical Partitioning in Object Oriented Databases

GAJANAN S. CHINCHWADKAR AND ANGELA GOH

School of Applied Science, Nanyang Technological University, Singapore 639798, Singapore
Email: Gajanan.Chinchwadkar@sybase.com

In this paper, some interesting issues related to vertical partitioning in object oriented database systems are presented. A review of existing research is given with an identification of some open problems. A taxonomy of various possible partitioning schemes and a unified view of the vertical partitioning problem are also presented. Existing vertical partitioning algorithms have been studied for their use in both parallel and distributed object-oriented databases.

Received April 8, 1997; revised January 11, 1999

1. INTRODUCTION

Database systems employ various partitioning techniques in order to achieve high resource utilization and performance. Most of the research related to partitioning has been carried out in the context of relational databases. Reported ways of partitioning a relation include vertical, horizontal and hybrid/mixed [1]. Vertical partitioning refers to the dissection of a relation into a set of relations, each containing a subset of attributes of the original relation. Horizontal partitioning refers to the dissection of a relation so that each smaller relation contains the same number of attributes as the original relation but with different tuples in each partition. Mixed partitioning refers to vertically (horizontally) partitioning a horizontal (vertical) partition.

An increasing demand on the performance of object oriented database systems (ODBSs) has resulted in the adoption of partitioning techniques from relational databases. In addition to vertical [2, 3, 4], horizontal [4, 5] and hybrid [5] partitioning, there are new ways specific to object-oriented (OO) databases such as path partitioning [4] and method-induced partitioning [6]. The OO features such as encapsulation, ISA/class-composition hierarchies and the presence of simple and complex methods add to the complexity of the partitioning problem. In this paper, we present issues related to the vertical partitioning of ODBSs. Vertical partitioning is inherently more difficult than horizontal partitioning because of its large solution space [1]. Vertical partitioning minimizes irrelevant IO and exploits the parallelism between the transactions which access parts of objects [3]. On the other hand, horizontal partitioning minimizes irrelevant IO by reducing the number of irrelevant objects accessed and exploits the parallelism between the transactions which access different objects of a class. With horizontal partitioning, methods are relatively easy to handle since they can be replicated in each partition. Vertical partitioning represents fragmentation of each object and hence it introduces interesting problems related to handling of methods. Another difference between vertical and horizontal partitioning is that the number of vertical partitions is bound by the number of attributes in a class, whereas in horizontal partitioning the number of partitions is bound by the number of objects in a class. This makes horizontal partitioning more scalable than vertical partitioning. Hence the former is more suitable for applications such as decision support systems and the latter more useful for on-line transaction processing (OLTP) systems where high transaction throughput is desirable by virtue of increased data availability [7].

In general, partitioning provides the following advantages [8] in distributed OO databases (DOBSs).

1. It reduces the amount of irrelevant data accessed by applications since applications usually access portions of classes.
2. It allows greater concurrency because 'lock granularity' can accurately reflect the applications using the object base.
3. It reduces the amount of data transferred when migration is required.
4. Partition replication is more efficient than replicating the entire class because it minimizes the update problem and the amount of storage utilized.

Similar advantages of vertical partitioning in parallel OO databases (POBSs) are presented in [9]. These include retrieval parallelism, an avoidance of irrelevant data access and an improvement in parallelism during updates since only specific partitions need to be locked. However, a drawback of partitioning of data is that the transactions may have common data requirements resulting in conflicting data accesses. In these situations, the formation of disjoint partitions is not possible. Hence the transactions that access multiple partitions suffer performance degradation [1]. The join costs become very high when the relation is vertically partitioned and the queries involve joins of many partitions. Similarly, for main memory databases such as PRACTIC

[7], the high join costs make vertical partitioning unsuitable.

Several fundamental issues related to partitioning an ODBSs have been reported in [5]. Our work differs in that it covers issues related to vertical partitioning in greater depth. It presents the issues which highlight the fact that ODBSs vertical partitioning problem is different from relation partitioning and reviews the reported research on the problem. It also identifies some open problems.

After this introduction, a review of the object oriented model and definitions adopted in this paper are given. This is followed by a discussion of vertical partitions within the OO framework and a taxonomy of various approaches to vertical partitioning. Method transformations are presented followed by a description of cost models and algorithms for vertical partitioning.

2. MODEL AND BASIC DEFINITIONS

The object model that is assumed in the subsequent discussion is briefly described in this section. There are variations in the object models used by different researchers. Hence, a core model is presented below and the differences are pointed out where necessary.

2.1. Object model

The OO data model supports basic features such as class, encapsulation, inheritance and unique object identifier (OID) [10]. Each object of a class has a *state* and *behaviour*. The state is represented by attributes and behaviour is represented by methods. Objects in a class (referred to as a subclass) can inherit attributes and methods from other classes (referred to as superclasses). The class subclass relationship represents an ISA hierarchy. Attributes can be either simple or complex. Simple attributes acquire values from an atomic domain, such as *integer* or *character*. Complex attributes can acquire values from the set of OIDs of objects in the database. Objects with complex attributes capture a 'is-part-of' relationship to form a class-composition hierarchy. The class corresponding to objects with complex attributes is referred to as a containing class. Classes corresponding to the OIDs assumed by the complex attributes are referred to as contained classes. Methods can be either simple or complex [5]. Simple methods cannot invoke other methods while complex methods can. These other methods may belong to the class in which the invoking method is bound or a superclass or a contained class or any class defined in the schema (similar to friend classes in C++).

2.2. Definitions

For any class $C(I, M)$, $I = \{a_1, a_2, \ldots, a_n\}$ is the set of attributes and $M = \{m_1, m_2, \ldots, m_q\}$ is the set of methods where n and q are positive integers. For each method $m_i \in M$, the set $\mathcal{A}m_i = \{a_{i_1}, \ldots, a_{i_k}\}$ represents the set of attributes accessed by the method m_i. It is referred to as the attribute view of method m_i.

DEFINITION 2.1. *$proj^A$ is a function which projects all the attributes in a class. For any class C, $proj^A : C \rightarrow I$, where I is the set of attributes of the class.*

DEFINITION 2.2. *ψ is a function which evaluates the ownership of a set of attributes $\mathcal{S}$. If all the attributes in $\mathcal{S}$ belong to the same class C, ψ evaluates to 1, and to 0 otherwise.*

$$\begin{aligned} \psi(\mathcal{S}) &= 1, \quad \mathcal{S} \subset proj^A(C), \\ &= 0, \quad otherwise. \end{aligned}$$

DEFINITION 2.3. *$\mathcal{P}$ is a function that maps $m_i \in M$ to a local method L_i of partition C_j or a global method G_i depending upon whether the method accesses attributes only from the partition C_j or different partitions.*

$$\begin{aligned} \mathcal{P}m_i &= L_i, \quad \psi(\mathcal{A}m_i) = 1, \\ &= G_i, \quad otherwise. \\ \mathcal{P}M &= \{\mathcal{P}m_i \mid m_i \in M\}. \end{aligned}$$

Several terms related to the information at the logical level of a database have been defined in [2, 3]. This information is obtained from *a priori* analysis of the database schema, its applications and access frequency of applications at different sites. We present a summary of these definitions below. We assume a query needs to access the attributes and/or methods of a class C.

1. A user query (application): a user query (or transaction) Q, accessing database objects is a sequence of method invocations on an object or set of objects.
2. Access frequency of user query (ACC): this is the number of invocations of a query per unit time:
$$\mathrm{ACC}(Q) = \frac{\text{No. of Invocations of } Q}{\text{Time}}$$
3. Method–method affinity (MMA): method–method affinity between two methods of the same class is the sum of access frequencies of all the user queries which access these two methods together:
$$\mathrm{MMA}(m_i, m_j) = \sum_{\forall Q \mid m_i \in \mathcal{M}Q \wedge m_j \in \mathcal{M}Q} \mathrm{ACC}(Q)$$
where $\mathcal{M}Q$ is the set of methods invoked by transaction Q.
4. Attribute–attribute affinity (AAA): attribute–attribute affinity between two attributes of the same class is the sum of access frequencies of all queries accessing these two attributes together:
$$\mathrm{AAA}(a_i, a_j) = \sum_{\forall Q \mid a_i \in \mathcal{A}Q \wedge a_j \in \mathcal{A}Q} \mathrm{ACC}(Q)$$
where $\mathcal{A}Q$ is the set of attributes accessed by methods in $\mathcal{M}Q$.
5. Attribute–method affinity (AMA): this is the sum of the access frequencies of all the queries accessing the attribute and the method together:
$$\mathrm{AMA}(a, m) = \sum_{\forall Q \mid a \in \mathcal{A}Q \wedge m \in \mathcal{M}Q} \mathrm{ACC}(Q)$$

6. Attribute–partition affinity (APA): suppose V is a subset of attributes, M is a subset of methods and $C_i(V, M)$ is the ith partition of class C. For an attribute a of class C such that $a \notin V$, the affinity between attribute a and partition C_i:

$$\mathrm{APA}(a, C_i) = \sum_{v \in V} \mathrm{AAA}(a, v) + \sum_{m \in M} \mathrm{AMA}(a, m).$$

7. Common method affinity (CMA): if a subset of attributes is accessed by a common subset of methods, then common method affinity between the two subsets is the sum of the AMA of each attribute with each method in the subset. Common attribute affinity (CAA) of a set of methods may be similarly defined.

3. REPRESENTATION OF VERTICAL PARTITIONS

This section describes the representation of vertical partitions within the OO framework and identifies the open problem of incorporating class–subclass hierarchy in this representation.

3.1. Single class partitions

As defined earlier, vertical partitions are subsets of attributes over all tuples in a relation. The vertical partitions of relational databases maintain closure. By closure we mean that any vertical partition is a relation in itself. In the case of ODBSs, it is more difficult to define vertical partitions because of the presence of methods in classes. Neither a subset of attributes alone nor a subset of methods alone, can form a vertical partition. This is because each partition must contain both attributes and methods. Further, in ODBSs, a strict adherence to the requirement of closure of partitioning implies that each subset must also be a class. Thus, the representation of the vertical partitions within the OO framework is an important issue in ODBSs.

A representation of vertical partitioning has been reported in [4]. In this representation, attributes are partitioned in the first place and relevant methods are inserted in the partitions afterwards. A partition is defined as a new class. Consider the example given in Figure 1. The class PERSON with three attributes and four methods is shown in Figure 1a. Two of the attributes, namely *Name* and *Address*, are simple. The third, *DateBirth*, is complex and hence it is marked with a '*' in the figure. The methods *m1*, *m2* and *m3* retrieve each of the three attributes respectively and the method *m4* returns age, given the name of a person. It should be noted that method *m4* must invoke a method in the contained class pointed to by the complex attribute DateBirth and that it accesses two attributes from the class PERSON, Name and DateBirth. Figures 1b and c show two different sets of partitions of class PERSON. A partition set consists of subsets of attributes and methods which represent the vertical partitioning of a given class. A unit of partitioning can be either a method or an attribute. Attribute (method) as a unit of partitioning implies attributes (methods) are partitioned first and methods (attributes) are included in the partitions subsequently. In the partition sets of Figures 1b and c, the unit is an attribute. Figure 1d shows a partition set in which the unit is a method. The partitioning schemes which use attribute and method as units may be referred to as the attribute-based partitioning scheme and the method-based partitioning scheme respectively. A partitioning algorithm is a procedure for obtaining a partition set. A partitioning scheme specifies the unit and the representation of partitions. A partition set may be viewed as an *instance* of a partitioning scheme.

User-defined methods in classes may access multiple attributes. Those methods which access attributes from only one partition are local to that partition and those accessing attributes from multiple partitions are global methods [4]. In Figure 1c, method *m4* is a global method. As shown in Figure 1b, another way of grouping the same attributes leads to the absence of global methods. It should be noted that both of these partition sets are obtained from attribute-based partitioning schemes. In relational databases, a partition set is the same as the partitioning scheme. Since a relation does not encapsulate methods, the concept of local and global methods does not exist.

A partition set using the scheme of [4] is illustrated in Figure 2. The class *PERSON′* is referred to as the partitioned equivalent class of the class PERSON. It is a composite class containing complex attributes which point to the individual partitions. It also contains global methods. This representation allows the global methods to access various partitions without violating encapsulation of the partitions. The class-composition hierarchy rooted at a partitioned equivalent class represents the logical equivalent of the non-partitioned class in the schema.

A partitioned database is said to be transparent (to applications) if the applications which are written for the non-partitioned database can execute against the partitioned database without any modifications. In other words, an application invokes the same set of methods irrespective of the database partitioning and the database system is responsible for correct execution of these methods on the partitioned database. It should be noted that by 'application', we mean queries that are written in declarative languages such as ObjectSQL. These queries are executed by invoking methods in classes in order to preserve encapsulation of objects. Since the global methods access attributes from different partitions and complex methods may invoke methods from other partitions, their execution paths change after partitioning. The mapping of calls from the original database to the partitioned one can be achieved through a preprocessing step called method transformation [6]. Method transformation involves replacing the attribute accesses in global methods and remote method invocations in the complex methods with appropriate path expressions.

It is possible to combine the basic partitioning schemes (vertical, horizontal and path) of [4] using information about different types of methods and the values returned by them. The resulting schemes can be referred to as *method-induced*

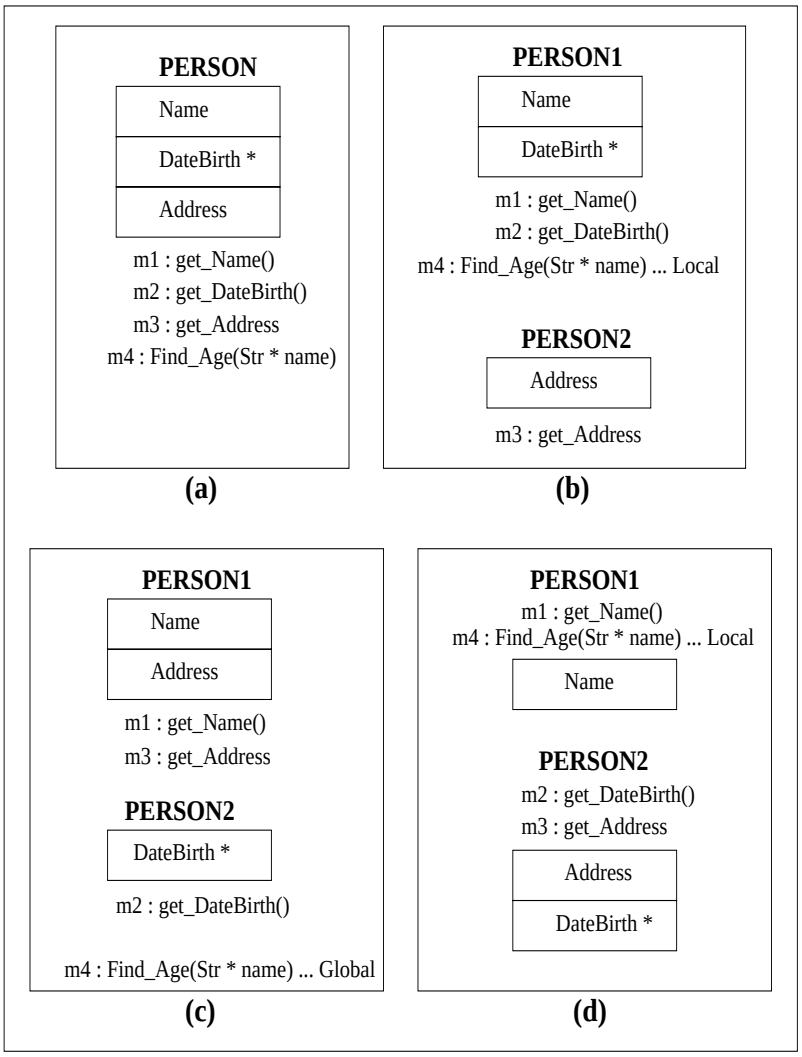

FIGURE 1. Various ways of partitioning a class.

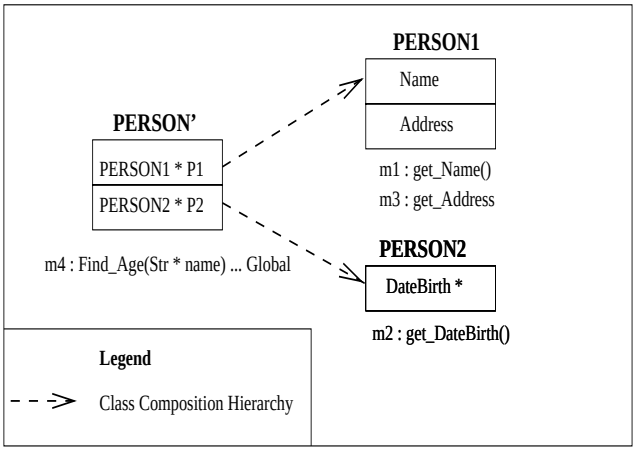

FIGURE 2. An attribute-based partitioning scheme.

partitioning schemes [6]. Complex methods invoking other methods form complex-method invocation paths. Method-induced partitioning schemes consider different complex-method invocation paths. Then, vertical partitions are formed by grouping attributes in different classes which are along the same complex-method invocation paths.

The vertical partitioning scheme [4] can be expressed as, $V = \{C'(I', M'), C_1(I^1, M^1), C_2(I^2, M^2), \ldots, C_p(I^p, M^p)\}$, where p represents the number of partitions. Class $C'(I', M')$ represents the root of the class-composition hierarchy which represents a logical equivalent of class $C(I, M)$ such that,

1. $I' = \{A_1, \ldots, A_p\}$. Each complex attribute A_j has a corresponding class $C_j(I^j, M^j)$, $1 \leq j \leq p$.
2. $M' = \{G_1, \ldots, G_g\}$, where g is a positive integer and $M' \subseteq M$. Each G_j from M' is a global method and accesses attributes from multiple classes. $G = \mathcal{P}m_i$, for some $m_i \in M$.
3. $I^j \subseteq I$ and $M^j \subseteq \mathcal{P}M$. Each class $C_j(I^j, M^j)$, where $1 \leq j \leq p$, is a vertical partition. Each $L \in M^j$ accesses attributes belonging to I^j and is referred to as a local method of the partition. $L = \mathcal{P}m_i$, for some $m_i \in M$.

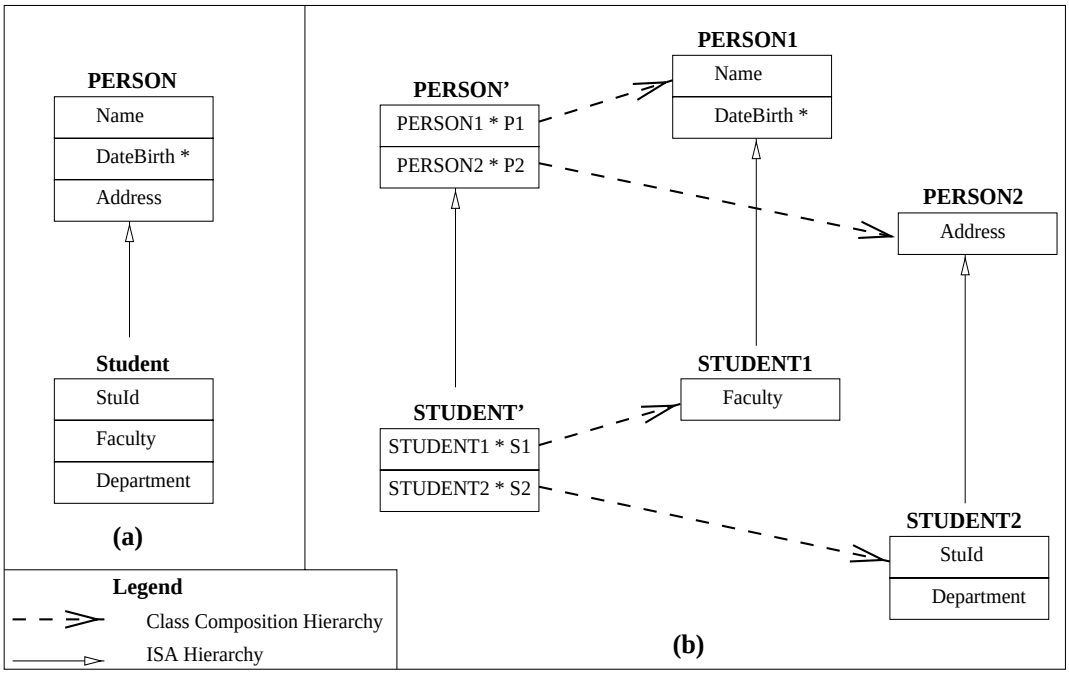

FIGURE 3. Presence of ISA hierarchy.

4. $I^j \cap I^k = \emptyset$, $M^j \cap M^k = \emptyset$ and $M' \cap M^j = \emptyset$, for $1 \leq j, k \leq p$ and $j \neq k$. In other words, the partitions do not overlap.
5. $\mathcal{P}M = M' \cup (\cup_{1 \leq j \leq p}\ M^j)$ and $I = \cup_{1 \leq j \leq p} I^j$. In other words, the set of partitions is complete.

3.2. The problem of the ISA hierarchy

Karlapalem and Li [4] have also proposed an extension to this scheme which considers the presence of an ISA hierarchy. With reference to Figure 3, the ISA hierarchy is represented by the PERSON–STUDENT class pair. STUDENT is a subclass of PERSON. Hence, each object of class STUDENT inherits all the attributes and methods in the class PERSON. If the class STUDENT is partitioned, there is a problem as to which attributes and methods should be inherited by each of its partitions in order to retain the logical equivalence with the original class, STUDENT. The scheme in Figure 3b shows that the partition STUDENT1 inherits all the attributes and methods of one of the partitions of its superclass, PERSON1. Similarly, STUDENT2 inherits the attributes and methods of PERSON2. Such a scheme is lossy; a complex method in STUDENT2 cannot invoke an inherited method which lies in the partition PERSON1 of the superclass and any method in STUDENT2 cannot access the attributes in the partition PERSON1. In order to overcome this problem, a modified scheme could have all the partitions of the subclass inheriting attributes and methods from all the partitions of the superclass. Although this scheme is better than the scheme in Figure 3b, it leads to several unresolved issues.

1. The global methods which belong to class PERSON′ are inaccessible through any of the partitions of the subclass STUDENT.
2. The aggregation of classes STUDENT′, STUDENT1 and STUDENT2 contains two instances of the methods and attributes of the partitions PERSON1 and PERSON2.
3. The global methods of class STUDENT′ cannot invoke the methods in the superclass which are local to the partitions PERSON1 and PERSON2.

Another possibility is that the partitioned equivalent class STUDENT′ should inherit the methods and attributes from classes PERSON′, PERSON1 and PERSON2. However, when queries access local methods of a partition, the cost of retrieval of partitioned-equivalent class may be higher than that of retrieving the original partition [11]. In these situations, a simple optimizing strategy would be to avoid storing instance objects of the partitioned-equivalent class explicitly. Instead, partition catalogues can be used to map method invocations made by the queries to the methods in the partitioned domain. Thus, capturing inheritance relationship information in the representation schemes is an open issue.

4. TAXONOMY

Based on the various reported partitioning schemes, a taxonomy of vertical partitioning schemes is proposed in this section. In addition to the reported schemes, the taxonomy covers all other possible schemes. Such a taxonomy is useful for developing partitioning algorithms and for studying existing research. Vertical partitioning algorithms utilize structural information and *logical* or *physical* (object size, number of objects and page size) information about the schema. It is possible to express the information used by an algorithm through the appropriate nomenclature of the

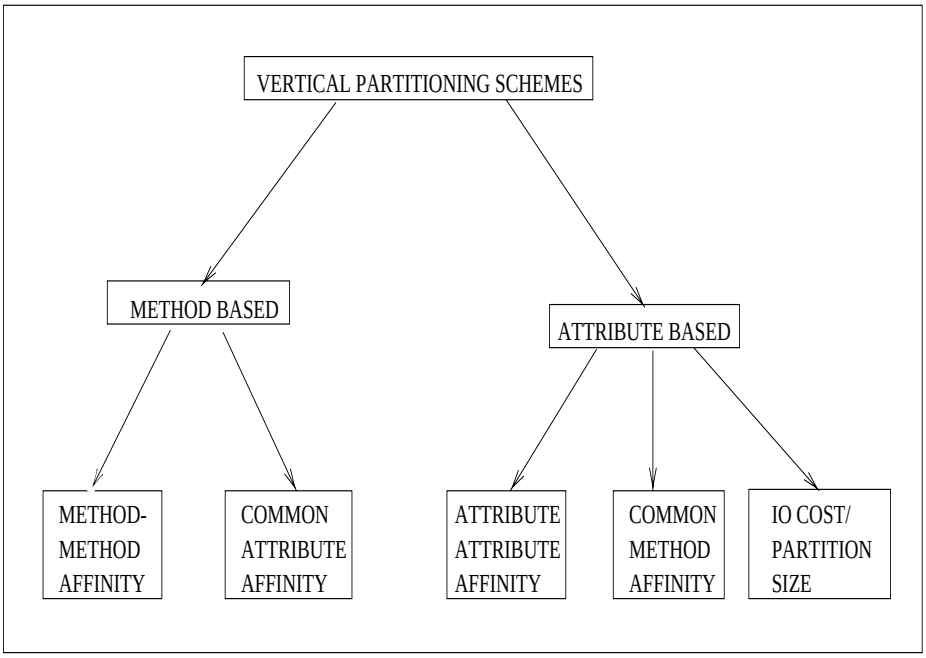

FIGURE 4. Taxonomy for vertical partitioning schemes for ODBSs.

associated scheme. Two basic forms of vertical partitioning scheme(s) (PS) are possible based purely on structural information of the class. They are method-based and attribute-based partitioning schemes. Each basic scheme can be further classified, depending upon the criterion for grouping methods or attributes. Figure 4 illustrates the classification of the schemes.

Method-based PS (MPS)

Methods in a class can be used as a basic unit for partitioning. In MPS, methods are partitioned in the first place and the attributes are inserted afterwards. MPS can be further classified as follows.

Method–method affinity (MMA)
Method–method affinity is used to select methods for grouping in this scheme. The methods can be grouped together based on either high or low affinities among themselves. The affinities could be estimated on the basis of transaction method access patterns over a predetermined set of transactions or on the basis of complex methods that invoke other methods. Grouping of methods with high affinity indicates that methods in any partition will be accessed together by the transaction. Grouping of methods with low affinity implies less competition among transactions for accessing any partition and more intraquery parallelism.

Common attribute affinity (CAA)
There is a possibility of grouping the methods which access the same subset of attributes in a partition along with the relevant attributes. Such a grouping can reduce data shipping in an environment where partitions are mapped onto different sites (processors). However, none of the reported algorithms are based on this scheme.

Theoretically one more class of MPS exists. This class is based on processing cost of methods in order to distribute the processing load among processors. Since the processing cost is found to be very small compared to IO cost in very large databases [12], such a scheme has no practical significance.

Attribute-based PS (APS)

In such a scheme, the attributes are partitioned first and methods are inserted afterwards. APS can be based on the following factors: logical level information about the database and the physical information about the database. The first two of the following subclasses are based on the logical level information while the third is based on the physical.

Attribute–attribute affinities (AAA)
This scheme groups attributes on the basis of affinities between the pairs of attributes. High AAA-based grouping implies low irrelevant data accesses in each partition and thereby fewer disk page accesses. However, this results in reduced data parallelism in single query processing. If queries access distinct portions of objects, high AAA schemes may result in increased data parallelism when different types of queries access the database concurrently. Low AAA-based clustering of attributes implies that the attributes in any partition are less likely to be accessed together over the predetermined set of transactions. This

ensures good distribution of IO load among the processors for single query processing but results in more irrelevant data accesses. It should be noted that applications access attributes through methods. Hence, analysis of method–attribute accesses is necessary for implementing this scheme.

Common method affinity (CMA)
In this case, attributes which are accessed by the same subset of methods are placed in one partition. These schemes can be useful when the schema contains more methods accessing disjoint subsets of attributes.

IO and communication cost
The sizes and number of attributes are directly related with partition size and IO cost in a database system. Hence attributes can be grouped together in partitions so as to distribute and balance the IO load. Partitioning also helps to minimize the communication load in a POBS, the IO and communication load in DOBSs and minimize the IO load in centralized ODBSs. An existing IO cost-based partitioning scheme [12] aims at minimizing the IO cost.

The degree of a partition refers to the number of attributes (methods) in the partition formed under an APS (MPS). In all these partitioning schemes, it is possible to form the partitions with fixed degrees or variable degrees. Both of the reported class partitioning algorithms [2, 3] are based on high MMA-based variable degree partitioning schemes. An example of the fixed degree vertical partitioning scheme with a unit degree can be found in [9]. The scheme proposed by [4] is an example of a general APS. An example of IO cost-based variable degree APS can be found in [12]. The traditional variable degree vertical partitioning has not been studied in the context of POBSs.

The use of MMA and AAA approaches has different implications in the partitioning algorithms. In the MMA approach, the aim is to distribute methods. A method distribution algorithm cannot guarantee data distribution. A given schema may contain more methods than the number of attributes because each attribute is associated with a retrieval method and additionally, other user-defined methods. As a result, the MMA matrix may become larger than the AAA matrix. Hence, the MMA approach can be expected to be computationally more intensive. It should be noted that the MMA matrices of [2, 3] also accommodate the access frequencies of transactions at different sites. The MMA-based approach may lead to a distribution of attributes in which some partitions are very narrow [2].

It should be noted that this taxonomy covers only the basic ways of formulating vertical partitions. It is possible to combine these basic approaches to form more efficient hybrid schemes which are not covered by this taxonomy. Some of the hybrid schemes are likely to be more useful in practice. An interesting study of comparing cost- and affinity-based approaches has been reported in [12]. The results show that a hybrid affinity cost-based approach is better than pure affinity-based approach.

5. METHOD TRANSFORMATIONS

The OO model allows two types of methods, namely, simple and complex. The partitioning scheme gives rise to two types of methods, local and global. This results in several types of methods in the partitioned domain, such as local/global simple methods, local/global complex methods invoking local/global simple methods and local/global complex methods invoking local/global complex methods. A detailed study of all these method types is necessary if database transparency is to be provided to the applications [13].

A unified view of different aspects of the class partitioning problem is shown in Figure 5. The partitioning processor is responsible for generation of partitions and transformation of methods. Input to the partitioning processor is a class of the user-defined schema. The output is the partitioned equivalent class with a partition set. The partitioning processor consists of two parts, the partitioning algorithm and the transformation processor. The partitioning algorithm forms partition sets and generates a partition catalogue. The transformation processor transforms the methods so that they can execute correctly in the partitioned domain. It relies upon the partitioning catalogue to determine whether a method is local or global and makes use of the transformation rules [13, 14]. For example, in Figure 2, the global method Find_Age(Str * name) refers to the attributes Name of PERSON1 and DateBirth of PERSON2. In the definition of this method, all the references to these two attributes should be replaced by appropriate path expressions, that is, $PERSON' \cdot P1 \cdot get\_Name()$ and $PERSON' \cdot P2 \cdot get\_DateBirth()$, respectively. It should be noted that due to encapsulation of objects, the remote attributes are retrieved by invoking retrieval methods in the remote objects. Transformations for the methods that create/destroy objects and read/write attributes have been presented in [6]. As an example assume that there is a constructor for class PERSON which accepts initial attribute values as arguments and returns a pointer to the newly created object. Further, assume that the complex attribute DateBirth points to an object of class DATE. Thus, creation of a new object for the scheme given in Figure 2 can be expressed as,

$$O = person(name, OID\text{-}DATE, address).$$

In the partitioned schema, three constructors *person′*, *person1* and *person2* are required. (This is slightly different from the example shown in [6] wherein a generic constructor which accepts class name as one of the arguments is assumed.) The constructor *person′* accepts two OIDs as arguments. These OIDs are generated using constructors of the partitions. The constructor for (partition) class PERSON1 accepts the values of two initializing attributes, *Name* and *Address*. The constructor for (partition) class PERSON2 accepts the value of initializing attribute, DateBirth. Thus, the creation of an object in the partitioned

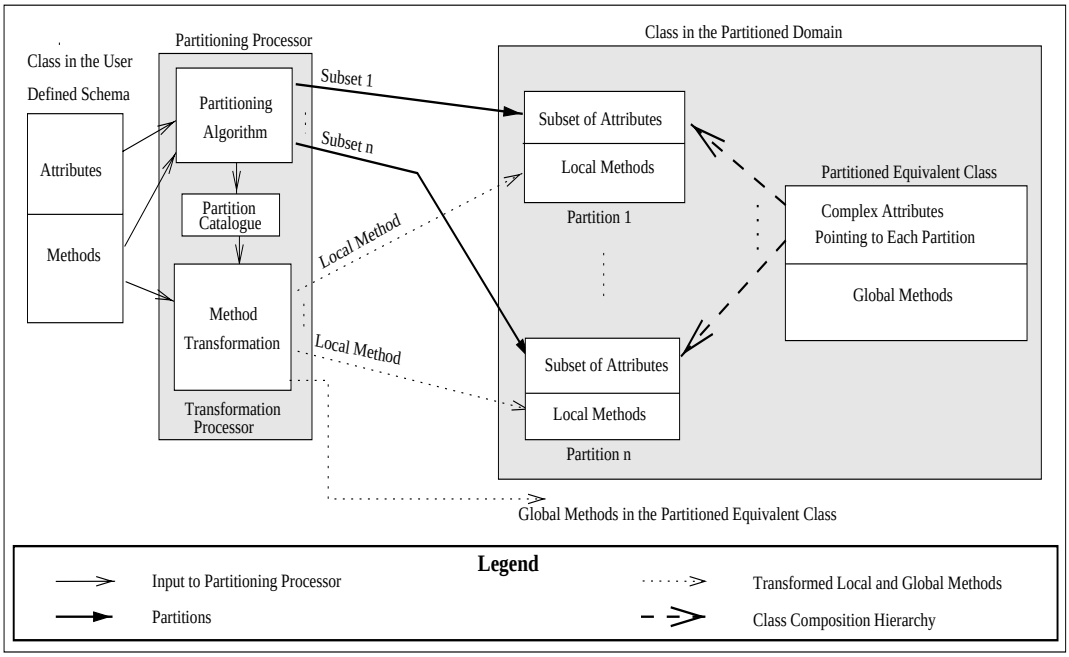

FIGURE 5. Unified view of the vertical partitioning problem.

domain can be expressed as,

$$O' = person'(person1(name, address), person2(OID\text{-}DATE)).$$

In the non-partitioned schema, the expression $O \cdot Name$ represents an access to attribute Name of the object O. Access to the same attribute in the partitioned schema is achieved by the path $O' \cdot P1 \cdot Name$. Similarly, the paths for the methods invoked by complex methods also undergo changes if the invoked methods belong to other partitions. The possible path traversal rules for transformations of different types of methods can be found in [13, 14]. In a partitioned but transparent ODBMS, the user applications/queries invoke the methods defined in the non-partitioned schema.

An open issue concerning transformations is whether to implement a static or run-time transformation processor. A static transformation processor becomes a part of the partitioning processor as shown in Figure 5. It performs transformations as a preprocessing step. Thus, the user-defined methods are stored in the preprocessed and transformed form in the partitioned database. In the case of a dynamic transformation processor, the query processor generates calls to the user-defined methods which are passed through a run-time transformation processor. Accesses to attributes and method invocations are dynamically replaced by appropriate path expressions. The advantages of the dynamic approach include the avoidance of costly preprocessing and transformation steps during database loading and the storage of user defined classes. However, it introduces run-time transformation overheads.

5.1. Correctness of vertical partitions

In order to ensure database consistency, some rules should be enforced on partitioning [1]. The rules for ensuring the correctness of relational database partitioning have been extended by [2, 8] for ODBSs.

1. Completeness: if a class is partitioned into different fragments, then every attribute or method in the class must be present in at least one of the fragments.
2. Reconstruction: if a class is decomposed into several fragments, it should be possible to define an operator ∇, such that by applying ∇ to the set of fragments, the original class can be reconstructed.
3. Disjointedness: if a class is partitioned into several fragments, then no attribute or method should belong to more than one fragment.

We believe that these conditions are not sufficient for correctness of ODBSs vertical partitioning. This is because in the process of partitioning of an object, the methods need to be transformed. Hence the transformations and reverse transformations during reconstruction also form essential requirements of correctness of vertical partitions. The requirement of disjointedness means that global methods may not be replicated. It is also necessary to transform the global methods. Thus, correctness of OO vertical partitioning requires consideration of the behavioural aspect of the OO model.

6. ALGORITHMS AND COST MODELS

Reported work on vertical partitioning of relations includes [15, 16, 17, 18, 19, 20]. The relation partitioning algorithms

which have influenced the class partitioning in ODBSs are the Navathe's vertical partitioning (NVP) algorithm [21] and the graphical algorithm [22]. These algorithms use *logical* level information obtained from *a priori* analysis of the database schema and transactions. The *a priori* analysis of data is feasible because of the assumption that 20% of the user queries account for 80% of the data accesses [23]. These algorithms work well for the database systems in which there are no drastic changes in the requirements of transactions [8]. Vertical partitions are defined as subsets of attributes in a relation. If the intersection set of the partition attributes is empty, the partitions are said to be *non-overlapping*, otherwise they are said to be *overlapping*. Throughout this paper, only non-overlapping partitions have been considered.

6.1. Algorithms

This section presents the NVP [21] and the graphical algorithm [22] for relation partitioning together with the reported ODBS vertical partitioning algorithms. Vertical partitioning algorithms work with different objectives. Some algorithms aim to group the attributes of a relation into partitions such that the attributes which are required together by most of the predefined transactions can be found in the same partition. Examples include the NVP algorithm [21] and the graphical algorithm of Navathe and Ra [22]. On the other hand, Cornell and Yu [15] model the vertical partitioning problem as an integer programming problem with an objective of minimizing the number of disk accesses.

Both the NVP and the graphical algorithms use *logical* level information obtained from *a priori* analysis of the database schema and transactions. Both algorithms are based upon AAA. They construct a data structure called the AAA matrix which is a square matrix representing the AAA between all pairs of attributes in a relation. NVP uses the bond energy algorithm (BEA) of [20] to rearrange the rows and columns in the AAA matrix such that the value of a global affinity function is maximized. The global affinity function measures the affinity between attributes in each column of the AAA matrix with the corresponding attributes in the adjacent columns on both sides. The rearranged matrix is called a clustered affinity (CA) matrix. This then becomes an input to the binary vertical partitioning algorithm (BVP). The BVP algorithm recursively partitions the CA matrix into two halves in order to minimize the number of transactions which access attributes in both the halves. Thus, the NVP is a two-phase algorithm which involves the BEA and BVP algorithms.

The graphical algorithm treats the AAA matrix as a complete weighted graph in which nodes represent attributes. The weight of any edge represents the affinity value between the attributes at the two ends of the edge. It then searches for affinity cycles in the graph such that the edges in any particular cycle have similar affinity values. Each time an affinity cycle is detected, the algorithm tries to expand it by considering inclusion of more edges with similar affinities. The graphical algorithm has advantages over the NVP algorithm in that it finds the partitions in one pass (there is no need of iterative/recursive execution of BVP) and it requires no objective function. Moreover, it is more efficient; the complexity of NVP is $O(n^2 \log n)$ and that of the graphical algorithm is $O(n^2)$ [22]. It should be noted that performance of a database partitioning algorithm cannot be stated only in terms of complexity. Since database partitioning is not performed frequently, the complexity analysis is a secondary issue.

In [24], the objectives of the BVP and the graphical algorithm are proved to be NP-hard and modifications to the graphical algorithm are proposed in order to obtain a polynomial time algorithm. Interestingly, this work has not been considered in the ODBS partitioning research.

Apart from relation partitioning algorithms, the other work which has influenced ODBS partitioning is the partition evaluator proposed by Chakravarthy *et al.* [25]. It is a function which consists of two components, namely, irrelevant local attribute access cost and relevant remote attribute access cost. In a distributed system, the first component is a measure of the local irrelevant data accesses and the second is a measure of communication with remote processors. This function can be used as an objective function in heuristic-based partitioning algorithms. It can also be used as a performance metric for existing algorithms.

A storage model for relational databases known as the 'decomposition storage model' (DSM) [26] resembles a vertical partition (at conceptual level) with one attribute. A relation is stored in terms of surrogate-attribute pairs under DSM. Advantages of DSM include simplicity, support for data models with multivalued attributes, entities, multiple parent relations and directed graphs. However, it occupies more storage space. DSM has been used in POBS research for vertically partitioning the database classes [9].

The use of information about logical affinities between attributes to construct the AAA matrix and subsequently using the NVP algorithm for partitioning classes has been suggested in [5]. The main difficulty in using this approach is determining the attribute accesses made by objects in the class composition and ISA hierarchies. To the best of our knowledge, there is no reported algorithm for APS schemes.

An algorithm using MMA matrix (high MMA based) is proposed by Ezeife and Barker in [3, 8]. This algorithm allows the OO model to support class composition and ISA hierarchies, simple and complex methods. The procedure for constructing the MMA matrix incorporates all the OO features. Affinity between any two methods is calculated on the basis of simultaneous invocations of the methods by applications. Suppose both these methods belong to class C. Then, the simultaneous accesses include direct access to the class C, accesses through subclasses of C, containing classes of C or other classes defined in the schema. After considering all these accesses, *a modified* MMA matrix is constructed and the NVP algorithm [21] is used to partition it. Attributes are inserted in each partition using information on attribute–partition affinities (APAs). The NVP algorithm was proposed for use with the AAA matrix. However, it is not sensitive to the manner in which the input matrix is

constructed. The matrix may be constructed using attribute affinities or method affinities.

Effectiveness of partitioning schemes can be measured in terms of how much better the given set of transactions performs after partitioning the database. Ezeife measured the effectiveness using the partition evaluator of Chakravarthy *et al.* [25] and proved the correctness of the algorithm using three properties of the partitions, namely, completeness, disjointedness and reconstructability [1]. The complexity of the Ezeife algorithm is $O(cqm + m^2 + fma)$, where f is the maximum number of fragments in a class, m is the maximum number of methods in a class, a is the maximum number of attributes in a class, c is the number of classes in the database and q is the number of applications (queries) accessing a database class. Thus, it is a polynomial time algorithm.

Another high MMA-based algorithm to partition OO databases is reported in Bellatreche *et al.* [2]. The Bellatreche algorithm also constructs the MMA matrix. Two main differences between algorithms of [2] and [3] are:

1. In the Ezeife OO model, complex methods can invoke the other methods from the same class, a superclass, a contained class or some other classes as defined in the schema. The Bellatreche model allows the complex methods to invoke only the methods in contained classes. Thus, the Ezeife OO model is more general.
2. After forming the method affinity matrix, the Bellatreche algorithm uses the graphical algorithm [22] to partition the MMA matrix treating it as a graph whereas the Ezeife algorithm uses the NVP algorithm.

Both of the high MMA-based algorithms are modified relation partitioning algorithms. They, therefore, have the same differences as the relation partitioning algorithms. For example, NVP is a two-phase algorithm and the graphical algorithm is one phase; NVP performs binary partitioning iteratively (or recursively) while the graphical algorithm is one pass; and by controlling the depth of iterations (recursion), it is possible to control the size of each partition in the NVP. The graphical algorithm provides no such control.

Partitioning of data is an important issue in improving the performance of parallel database systems [27]. The vertical partitioning schemes used in the parallel ODBS research are based upon the DSM [28, 29]. There are no reports of the use of the variable degree partitioning scheme in these systems.

Algorithms for processing OO database queries in parallel have been proposed in [9]. This work reports the use of vertical partitioning where each class is partitioned into (OID, attribute) pairs and all partitions corresponding to a class are mapped onto different processors. Such a partitioning scheme is different from traditional vertical partitioning schemes in DOBS in that it uses fixed (and unit) degree partitions and does not consider the effect of complex methods, global methods and various hierarchies.

Haddleton [28] used a mixed partitioning scheme in which subsets of objects from a class are mapped onto different nodes and each node stores the subset using the DSM model. It is assumed that the individual fields in the objects may be large in size. For large size fields, DSM reduces the cost of dropping/adding fields to the object schema and the irrelevant attribute accesses.

We briefly examine the possibility of using the existing DOBS partitioning algorithms in parallel systems. Both the Ezeife and Bellatreche algorithms use method as a unit of partitioning. Naturally, a method distribution algorithm cannot guarantee a good attribute/data distribution (we refer to the example shown in [2]). This affects the distribution of IO in POBS. Alternatively, if a high AAA-based algorithm is developed, it will generate the partitions with the objective of minimizing the irrelevant IO. The attributes which are collectively used by most of the applications form a single partition. This, however, results in reduced IO parallelism. Low AAA-based schemes can increase the IO parallelism. The drawback is that the irrelevant data accesses also correspondingly increase. Thus, both the high and low AAA-based approaches are unsuitable for parallel ODBMS.

A cost-based algorithm suitable for an asynchronous shared nothing model has been proposed in [11]. This algorithm uses simulated annealing, a randomized hill climbing technique with an objective of minimizing a cost function. The cost function represents a combination of parallel IO time and overheads of communication arising out of vertical partitioning.

6.2. Cost model and communication overheads

The effectiveness of a partition set can be analysed using a cost model. In a distributed system, a cost model should take into account the IO performed at each site and intersite communication. In a centralized system, only IO is taken into account. Recently, a cost model for executing queries against a vertically partitioned OO database has been proposed by Fung *et al.* [12]. It measures the IO cost when a set of queries is executed against a vertically partitioned OO database. The IO cost consists of three components.

1. The cost of loading partitions of the root class of a class-composition hierarchy in the main memory.
2. The cost of loading the partitions of other classes in the class-composition hierarchy for predicate evaluation. If any class in the class-composition hierarchy is not partitioned, the cost of loading the entire class is considered.
3. The cost of loading appropriate partitions which contain the attributes required in the result of the query.

The cost of loading partitions of the classes in the class-composition hierarchy is computed using Yao's estimate [30]. This computation enforces an amount of sequentiality in the IO activity because it considers the IO for loading only those objects of a class which are referenced by the objects of the predecessor class in the class-composition hierarchy. Hence the use of this cost model is restricted to synchronous distributed systems or centralized systems. A cost function can also be used as an objective function in an algorithm. Fung *et al.* [12] have studied the goodness of the

best partitions formed using the cost-based approach and the affinity-based approach and concluded that the cost-based approach outperforms its affinity-based counterpart.

Equations for parametrizing the IO in a vertically partitioned POBS can be found in [9]. These equations have been used for simulating a database for the performance evaluation of parallel query processing algorithms. However, these equations do not estimate communication cost and the effect of complex methods on the communication and IO cost.

A cost model that estimates the gains in parallel IO time and the communication overheads that are introduced by vertical partitioning has been presented in [11]. This cost model assumes shared nothing architecture and it incorporates both complex and global methods. Assuming that each partition is mapped onto disks of different processing nodes the communication overheads incorporated by this model are:

1. A global method accesses data from multiple partitions. Therefore, there is a cost of requesting the data and also shipping it to the appropriate processor.
2. If a complex method invokes another method in its own class, vertical partitioning may result in placing these two methods in different partitions. Consequently, invoking the remote method and obtaining results from the remote method introduce communication overheads after partitioning.

Open problems in the cost analysis of vertical partitioning include a consideration of communication cost in DOBSs and extending the existing cost models or developing new models to incorporate shared-disk and shared-everything architectures in the context of POBSs. There is also a need to study the combined gain due to vertical partitioning and indexing. Vertical partitioning and indexing are two orthogonal techniques to avoid irrelevant IO [12].

7. CONCLUSIONS

Vertical partitioning is a well known technique in DOBSs which has also been employed in POBSs. In relational databases, it has been shown that vertical partitioning is useful even in centralized systems [21]. Similarly, it can also be useful in centralized ODBSs for reducing irrelevant IO and exploiting the concurrency between applications accessing different parts of an object.

In this paper we have defined basic terms such as partition sets, partitioning schemes and partitioning algorithms and have reviewed the literature relating to the vertical partitioning problem. We have considered both distributed and parallel ODBSs. A taxonomy for vertical partitioning schemes and several open problems have been presented. In both POBSs and DOBSs, effective partitioning schemes, algorithms, transformation processors and cost models need to be developed. We believe that the unified presentation of the vertical partitioning problem in this paper will facilitate further research on the problem.

ACKNOWLEDGEMENTS

Some of the papers referred to in this work were made available by Dr Christiana Ezeife (University of Windsor, Canada), Ladjel Bellatreche (Laboratoire TIMC-IMAG, France) and Dr Kamalakar Karlapalem (The Hong Kong University of Science and Technology). They have also helped us to gain a clearer understanding of their respective work and we acknowledge them for this. We also acknowledge Dr Ying Huang for making available some of the publications of the University of Florida Database Research Group and to Dr Russel Haddleton, University of Virginia, for the explanation of partitioning scheme used in ADAMS. Special thanks go to the anonymous reviewers for the valuable comments about the POBS partitioning work and the partition representation in the presence of ISA hierarchy.

REFERENCES

[1] Tamer Ozsu, M. and Valduriez, P. (1991) *Principles of Distributed Database Systems*. Prentice-Hall, Englewood Cliffs, NJ.

[2] Bellatreche, L., Simonet, A. and Simonet, M. (1996) Vertical fragmentation in distributed object database systems with complex attributes and methods. In *Proc. 7th Int. Conf. and Workshop on Database and Expert Systems and Applications (DEXA'96)*, Zurich, pp. 15–21.

[3] Ezeife, C. I. and Barker, K. (1995) Vertical fragmentation for advanced object models in a distributed object based system. In *Proc. 8th Int. Conf. on Computing and Information*. IEEE Publishers.

[4] Karlapalem, K. and Li, Q. (1995) Partitioning schemes for object oriented databases. In *Proc. 5th Int. Workshop on Research Issues in Data Engineering—Distributed Object Management (RIDE-DOM'95)*, Taipei, pp. 42–59.

[5] Karlapalem, K., Navathe, S. B. and Morsi, M. M. A. (1994) Issues in distribution design of object oriented databases. In *Distributed Object Management*, Tamer Ozsu, M. *et al.* (eds), Morgan Kaufman Publishers, San Mateo, pp. 148–164.

[6] Karlapalem, K., Li, Q. and Vieweg, S. (1996) Method induced partitioning schemes in object oriented databases. In *Proc. 16th Int. Conf. on Distributed Computing Systems*, Hong Kong, pp. 377–384.

[7] Bassiliades, N. and Vlahavas, I. (1996) Hierarchical query execution in a parallel object-oriented database system. *Parallel Computing*, **22**, 1017–1048.

[8] Ezeife, C. I. (1995) *Class Fragmentation in a Distributed Object Based System*. PhD Thesis, Department of Computer Science, University of Manitoba, Canada.

[9] Thakore, A. K. and Su, S. Y. W. (1994) Performance analysis of parallel object-oriented query processing algorithms. *J. Distributed Parallel Databases*, **2**, 59–100.

[10] Kim, W. (1990) *Introduction to Object-Oriented Databases*. MIT Press, Cambridge, MA.

[11] Chinchwadkar, G. S. (1997) *Vertical Partitioning in Object Oriented Databases*. PhD Thesis, Nanyang Technological University, Singapore.

[12] Fung, C. W., Karlapalem, K. and Li, Q. (1997) Cost-driven evaluation of vertical class partitioning in object oriented databases. In *Proc. 5th Int. on Database Systems for Advanced Applications (DASFA'97)*, Melbourne, pp. 11–20.

[13] Chinchwadkar, G. S. and Goh, A. (1997) Method transformations for vertical partitioning in parallel and distributed object databases. In *Proc. European Parallel Processing Conf. (EuroPar'97)*, Passau, pp. 1135–1143.

[14] Chinchwadkar, G. S. and Goh, A. (1997) Transforming complex methods in vertically partitioned OO databases. In *Proc. IASTED Int. Conf. on Parallel and Distributed Computing and Networks*, Singapore, pp. 56–59.

[15] Cornell, D. and Yu, P. S. (1987) A vertical partitioning algorithm for relational databases. In *Proc. 3rd Int. Conf. on Data Engineering*, Los Angeles, pp. 30–35.

[16] Chu, W. C. and Ieong, I. O. (1993) A transaction based approach to partitioning for relational database systems. *IEEE Trans. Software Eng.*, **19**, 804–812.

[17] Hammer, M. and Niamir, B. (1979) A heuristic approach to attribute partitioning. In *Proc. ACM SIGMOD*, Boston, pp. 93–101.

[18] Hoffer, J. A. and Severance, D. G. (1975) The use of cluster analysis in physical database design. In *Proc. 1st Int. Conf. on Very Large Databases*, Framingham, pp. 69–86.

[19] Huang, Y.-F. and Van, C.-H. (1995) Vertical partitioning in database design. *Information Sciences*, **86**, 19–35.

[20] McCormick, W. T., Schweitzer, P. J. and White, T. W. (1972) Problem decomposition and data reorganisation by a clustering technique. *Operations Research*, **20**, 993–1009.

[21] Navathe, S. B., Ceri, S., Wiederhold, G. and Dou, J. (1984) Vertical partitioning algorithms for database design. *ACM Trans. on Database Systems*, **9**, 680–710.

[22] Navathe, S. B. and Ra, M. (1989) Vertical partitioning for database design: a graphical algorithm. In *Proc. 1989 ACM SIGMOD Conf.*, Portland, pp. 440–450.

[23] Widerhold, G. (1982) *Database Design*. McGraw-Hill, New York.

[24] Lin, X., Orlowska, M. and Zhang, Y. (1993) A graph based cluster approach for vertical partitioning in database design. *Data Knowledge Eng.*, **11**, 151–169.

[25] Chakravarthy, S., Muthuraj, J., Varadrajan, R. and Navathe, S. B. (1992) *An Objective Function for Vertically Partitioning Relations in Distributed Database and its Analysis*. Technical Report UF-CIS-TR-92-045, Department of Computer and Information Sciences, University of Florida, p. 25.

[26] Copeland, G. P. and Khoshafian, S. N. (1985) A decomposition storage model. In *Proc. 1985 ACM SIGMOD Conf.*, Austin, pp. 268–279.

[27] Gray, J. P. and Jelly, I. E. (1991) Object-oriented approach for transputer based database system. *Information and Software Technol.*, **33**, 31–37.

[28] Haddleton, R. (1995) *An Implementation of a Parallel Object Oriented Database System*. Technical Report CS-95-49, Computer Science Deptartment, University of Virginia.

[29] Thakore, A. K. and Su, S. Y. W. (1993) Greedy heuristic mapping of object-oriented semantic schemes onto nodes of a regularly homogeneously connected parallel architecture. In *Proc. 1993 Object-Oriented Simulation Conf.*, La Jolla, CA.

[30] Yao, S. B. (1977) Approximating block accesses in database organizations. *Commun. ACM*, **20**, 260–261.