Horizontal Class Partitioning in Object-Oriented Databases*

Ladjel Bellatreche[1] and Kamalakar Karlapalem[1] and Ana Simonet[2]

[1] University of Science and Technology
Department of Computer Science
Clear Water Bay Kowloon, Hong Kong
email : {ladjel, kamal}@cs.ust.hk
[2] TIMC-IMAG Laboratory
Faculty of Medicine La Tronche 38706 France
email : asimonet@imag.fr

Abstract. The Horizontal Fragmentation (HF) is a process for reducing the number of disk access to execute a query by reducing the number of irrelevant objects accessed. In this paper, we present horizontal fragmentation based on a set of queries, and develop strategies for two versions of HF: *primary* and *derived*. Primary horizontal fragmentation of a class is performed using predicates of queries accessing this class. Derived horizontal fragmentation of a class is the partitioning of a class based on the horizontal fragmentation of another class.

1 Introduction

The methodology for distributed database design consists of data fragmentation and data allocation. Data fragmentation is the process of clustering data from a class by grouping relevant attributes and objects accessed by an application into class fragments. Data fragmentation enhances performance of applications as it reduces the amount of irrelevant data to be accessed and transferred among different sites in a distributed system. The decomposition of a class into class fragments also permits concurrent processing since a query can access fragments of a same class simultaneously [12]. The problem of data fragmentation in the distributed relational database systems has been recognized for its impact upon the performance of the system as a whole [6], [15] and [16]. The object-oriented database (OODB) environment supports an object-oriented data model which is built around the fundamental concept of an object, and includes features such as encapsulation, inheritance, class composition hierarchy, etc., complicate the definition of horizontal class fragmentation, and then we can not apply the techniques used in the relational model [15] and [16] in a straightforward manner. In the object-oriented environment, the fragmentation is a process which breaks a class into a set of class fragments. A class can be fragmented horizontally or vertically. A horizontal class-fragment is a nonempty proper subset of objects, while a vertical class-fragment is defined by a nonempty proper subset of attributes [3].

* Paper accepted in DEXA'97 Conference

1.1 Related Work

The issues involved in distributed design for an object database system are presented in [11], and the authors suggested that the techniques used in relational model [13] can be extended to object databases. However, they do not present an *effective* solution. In [9, 10], the authors developed representation schemes for horizontal fragmentation, and presented a solution for supporting method transparency in OODBs. Ezeife et al. [8] presented a set of algorithms for horizontally fragmenting the four class models: classes with simple attributes and methods, classes with complex attributes and simple methods, classes with simple attributes and complex methods and classes with complex attributes and methods. They used the algorithm developed by [16]. However, the number of minterms [16] generated by this algorithm is exponential to the number of predicates. The authors do not discuss the derived horizontal class fragmentation object-oriented environment. Bellatreche et al. [3] studied the vertical fragmentation problem for a model of class with complex attributes and methods.

1.2 Data Model

In this paper, we use an object-oriented model with the basic features described in the literature [1, 5]. Objects are uniquely identified by object identifiers (OID). Objects having the same attributes and methods are grouped into a class. An instance of a class is an object with an OID which has a set of values for its attributes. Classes are organized into an inheritance hierarchy by using the specialization property (isa), in which a subclass inherits the attributes and methods defined in the superclass(es). The database contains a root class which is an ancestor of every other class in the database. Two types of attributes are possible(simple and complex) : a simple attribute can only have an atomic domain (e.g., integer, string). A complex attribute has a database class as its domain. Thus, there is a hierarchy which arises from the aggregation relationship between the classes and their attributes. This hierarchy is known as class composition hierarchy which is a rooted directed graph (RDG) where the nodes are the classes, and an arc between pair of classes C_1 and C_2, if C_2 is the domain of an attribute of C_1. An example of such a RDG is given in figure 1. The methods have an signature including the method's name, a list of parameters, and a list of return values which can be an atomic value (integer, string) or an object identifier (OID).
The main contributions of this paper are:
1. Study of the impact of queries on the horizontal fragmentation.
2. Development of two algorithms (primary and derived) to achieve the horizontal class fragmentation.
3. Confirmation of the conjecture that some fragmentation techniques presented for the relational approach can be generalized and applied for the object-oriented model [11].

The rest of paper is organized as follows : section 2 presents some definitions used in specifying the problem of horizontal fragmentation, section 3 presents

the primary algorithm, section 4 introduces the derived algorithm, and section 5 presents conclusions.

2 Basic Concepts

Before describing the horizontal class fragmentation algorithm, some definitions are presented.
Notation: If a_i is an attribute of the class C_i, then:
• $a_i : C_i$ denotes a single valued attribute.
• $a_i : Const(C_i)$ denotes a multi-valued attribute.

Definition 1. A query in object-oriented database models has the following structure [5]: q ={Target clause; Range clause; Qualification clause}.
1) *Target clause* specifies some of the attributes of an object or the complete object of the class that is returned. That is, v or $v.a_i$, where v denotes the complete object, and $v.a_i$ denotes attribute a_i of object v.
2) *Range clause* contains the declaration of all object variables that are used in the qualification clause. It is denoted as: v/C where C is a class.
3) *Qualification* clause defines a boolean combination of predicates by using the logical connectives: $\wedge$, $\vee$, $\neg$. We denote the cardinality of a qualification clause $|qc|$ as the number of simple predicates it contains.

Example 1. The query q for retrieving the name of all projects whose cost is greater than \$70000 and located at "Hong Kong" is formulated as:
q = {$v.Pname$; $v/$ Project; $v.Cost() >$ \$70000 $\wedge$ $v.Location =$ "Hong Kong"}.

Definition 2. (Simple predicate) Banerjee et al. [2] define a *simple predicate* as a predicate on a simple attribute and it is defined as :
attribute_name $<$ *operator* $>$ *value*, where operator is a comparison operator ($=, <, \leq, >, \geq, \neq$) or a set operator ($contained-in, contains, set-equality, etc.$). The value is chosen from the domain of the attribute.

Since some object-oriented systems, such as Orion, and extensions of relational systems such as PostQuel, allow usage of methods in queries [5], we extend the definition of simple predicate defined by [2] to:
Attr_Meth $<$ *operator* $>$ *value*, where Attr_Meth is an attribute or a method, operator is as defined obove. The *value* is chosen from the domain of the attribute or the value returned by a method. A query that involves only simple predicates will be called *simple query*, such as {Cost () > \$70000}.

Definition 3. A path P represents a branch in a class composition hierarchy and it is specified by $C_1.A_1.A_2....A_n$ (n ≥ 1) where :
• C_1 is a class in the database schema
• A_1 is an attribute of class C_1
• A_i is an attribute of class C_i such that C_i is the domain of the attribute A_{i-1}

of class C_{i-1}, $(1 < i \leq n)$. For the last class in the path C_n, you can either access an attribute A_n, or a method of this class which returns a value or set of OIDs. The length of the path P is defined by the number of attributes, n, in P. We call the last attribute/method A_n of P the *sink* of the path P.

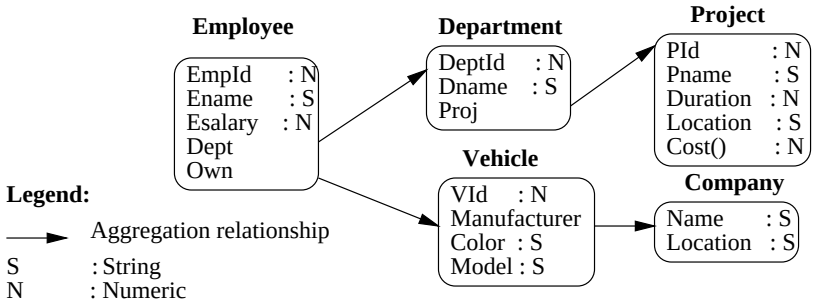

Fig. 1. The Class Composition Hierarchy of the class Employee

Definition 4. (Component predicate) A component predicate p_j $(1 \leq j \leq n)$ is a predicate defined on a path. A query that involves component predicate(s) will be called a *component query*.

3 Primary Horizontal Class Partitioning Algorithm

Horizontal Class Partitioning (HCP) varies according to the type of queries (simple or component). We will first discuss HCP for simple queries(primary horizontal class fragmentation), and then HCP for component queries(derived horizontal class fragmentation).

We assume that the class C_i to be fragmented. Let $Q = \{q_1, q_2, ..., q_l\}$ be a set of queries defined on class C_i, and each query q_j $(1 \leq j \leq l)$ is executed with a certain frequency. There are two types of queries that can be defined on a class C_i: simple queries S and component queries E, note $S \cup E = Q$. These component queries are defined on other classes in the class composition hierarchy but they have attributes or methods of C_i as their sinks. For example, the predicate Pname = "Database" is a simple predicate for the class *Project*, where as it is a component predicate for classes *Department* and *Employee*.

Primary Algorithm

Step 0 : Determine the predicates Pr used by queries defined on the class C. These predicates are defined on set of attributes A and the set of methods M. Let $|Pr| = n$, $|A| = r$ and $|M| = m$ represent the cardinality of Pr, A and M, respectively.

Step 1 : Build the predicate usage matrix of the class C. This matrix contains queries as rows and predicates as columns. The value(q_i, p_j) of this matrix equals 1 if a query q_i uses a predicate p_j; else it is 0.

Step 2 : Construct the predicate affinity matrix which is square and symmetric, where each value $aff(p_i, p_j)$ can be : 1) Numerical, representing the sum of the frequencies of queries which access simultaneously p_i and $p_j (1 \leq i, j \leq n)$, 2)

Non numerical, where the value " $\Rightarrow$ " of $aff(p_i, p_j)$ means that predicate p_i implies predicate p_j, value " $\Leftarrow$ " means that predicate p_j implies predicate p_i, and the value "$*$" means that two predicates p_i and p_j are "similar" introduced in [14] in that both are used jointly with a predicate p_l.
Step 3 : In this step, we apply the algorithm described in [14] to group the predicates to form clusters where the predicates in each cluster demonstrate high affinity to one another. The partitions of the graph generates a set of subsets $\mathcal{P} = \{P_1, P_2, ..., P_s\}$ of predicates.
Step 4 : In this step, we optimize the predicates contained in each subset by using predicate implication. We obtain a set of subsets $\mathcal{P}' = \{P'_1, P'_2, ..., P'_\alpha\}$ $(\alpha \leq s)$ of optimized predicates.
Step 5 : For each subset P'_i of $\mathcal{P}'$ resulting from step 4, we enumerate the attributes and the methods $Attr\_Meth$ not used by P'_i. For an $Attr\_Meth_j$ of $Attr\_Meth$, let w_j be the number of predicates defined on it $\{p_{i1}, p_{i2}, ..., p_{iw_j}\}$. We split P'_i into w_j subsets P'_{ik} $(1 \leq k \leq w_j)$, where each P_{ik} contains the predicate(s) of P'_i plus p_{ik}. We repeat this step until each subset P'_i uses all the attributes and methods. From the set $\mathcal{P}$, we build the attribute/method usage matrix which is an $(\alpha * (r + m))$ matrix and contains the subsets of predicates as rows and attributes A and methods M as columns. Each value (i, j) of this matrix is equal to 1 if $Attr\_Meth_j$ is used by a predicate of P'_i; otherwise it is 0.
Step 6 : If the predicates in each subset refer to the same attribute or method we link them by an OR connector, otherwise we use an AND connector to generate the class fragments. The final number of horizontal fragments will be equal to the number of subsets obtained step 4 plus one, including the fragment defined by the negation of the disjunction of all predicates previously defined, that we call ELSE.
Step 7 : Our algorithm may give rise to overlapping fragments. This step consists of refining these fragments in order to obtain non overlapping fragments.

Example 2. In Figure 1, we give the class composition hierarchy of a class *Employee*. The class Employee has two complex attributes: Dept, and Own, and three simple attributes: EmpId, Ename and Esalary which represent respectively the identifier, the name and the salary of an employee. We assume that the class *Project* will be fragmented and let the set of queries defined on this class:
$q1$:{v.PId; v/ Project; v.Duration $\leq 3 \wedge$ v.Cost() > 7000}.
$q2$:{v.Pname; v/ Project; v.Duration $\leq 4 \wedge$ v.Cost() > 7000}.
$q3$:{v.Dname; v/ Department; v.Proj.Duration $= 2 \wedge$ v.Proj.Cost() $\leq 7000 \wedge$ v.Proj.Location = "Hong Kong" }.
$q4$:{v.Esalary; v/ Employee; $5 \leq v$.Dept.Proj.Duration $\leq 6 \wedge v$.Dept.Proj.Cost() ≤ 7000 }.
We notice that q_1 and q_2 are simple and q_3 and q_4 are component and their sinks are in the class *Project*. The predicate usage matrix of the *Project* class is shown in figure 2. Let $p_1, p_2, ..., p_7$ the predicates used by the queries q_1, q_2, q_3, q_4 be: p_1 : Duration ≤ 3, p_2 : Duration ≤ 4 , p_3 : Duration $= 2$, p_4 : $5 \leq$ Duration ≤ 6 , p_5 : Cost() > 7000, p_6 : Cost() ≤ 7000, p_7 : Location = "Hong Kong". The attributes used by these predicates are, Duration and Location which will

be renamed by a_1 and a_2, respectively, and there is one method used by these predicate: $Cost()$ which will renamed as m. We notice that: $p_1 \Rightarrow p_2$, $p_3 \Rightarrow p_1$, $p_3 \Rightarrow p_2$, p_1 and p_2 are similar, and p_3 and p_4 are similar. We also add the access frequency column which shows the number of accesses to a predicate for a specific period for each query as shown figure 2.

$$\begin{array}{c} \\ q_1 \\ q_2 \\ q_3 \\ q_4 \end{array} \begin{array}{c} \begin{array}{cccccccc} p_1 & p_2 & p_3 & p_4 & p_5 & p_6 & p_7 & acc \end{array} \\ \begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 0 & 0 & 20 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 35 \\ 0 & 0 & 1 & 0 & 0 & 1 & 1 & 30 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 & 15 \end{pmatrix} \end{array}$$

Fig. 2. Predicate Usage Matrix

The value associated with two predicates in figure 3 represents the sum of the frequencies of the queries which access simultaneously these two predicates.

$$\begin{array}{c} \\ p_1 \\ p_2 \\ p_3 \\ p_4 \\ p_5 \\ p_6 \\ p_7 \end{array} \begin{array}{c} \begin{array}{ccccccc} p_1 & p_2 & p_3 & p_4 & p_5 & p_6 & p_7 \end{array} \\ \begin{pmatrix} 20 & \Rightarrow, * & \Leftarrow & 0 & 20 & 0 & 0 \\ \Leftarrow, * & 35 & \Leftarrow & 0 & 35 & 0 & 0 \\ \Rightarrow & \Rightarrow & 30 & * & 0 & 30 & 30 \\ 0 & 0 & * & 15 & 0 & 15 & 0 \\ 20 & 35 & 0 & 0 & 55 & 0 & 0 \\ 0 & 0 & 30 & 15 & 0 & 45 & 30 \\ 0 & 0 & 30 & 0 & 0 & 30 & 30 \end{pmatrix} \end{array}$$

Fig. 3. Predicate Affinity Matrix

We now apply step 3 of the PA to this matrix. Figure 4 shows three partitions of predicates $P_1 = \{p_1, p_2, p_5\}$, $P_2 = \{p_3, p_6, p_7\}$, $P_3 = \{p_4\}$.

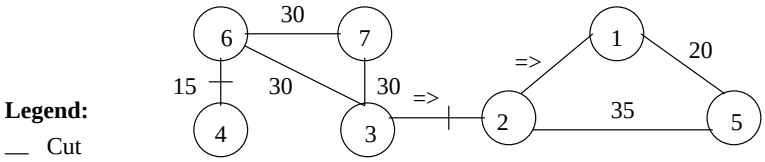

Fig. 4. Predicate Sets Generated by Primary Algorithm

The subset P_1 will be refined into $P_1' = \{p_2, p_5\}$ because $p_1 \Rightarrow p_2$. After the optimization step, we obtain three subsets: $P_1 = \{p_2, p_5\}$, $P_2 = \{p_3, p_6, p_7\}$, $P_3 = \{p_4\}$.

$$\begin{array}{c} \\ s_1 \\ s_2 \\ s_3 \end{array} \begin{array}{c} \begin{array}{ccc} a_1 & a_2 & m \end{array} \\ \begin{pmatrix} 1 & 0 & 1 \\ 1 & 1 & 1 \\ 1 & 0 & 0 \end{pmatrix} \end{array}$$

Fig. 5. Attribute/Method Usage Matrix.

In the attribute/method usage matrix in figure 5, we remark that the subset P_2 uses all the attributes and methods, but the subset P_1 does not use the attribute a_2, we notice that there is one predicate defined on a_2 (Location = "Hong Kong"), therefore, we split P_1 into P_{11} (see Step 5 of PA) where: $P_{11} = \{p_2, p_5, p_7\}$. Similarly, the subset P_3 will be split into:
$P_{31} = \{p_4, p_5, p_7\}$, $P_{32} = \{p_4, p_6, p_7\}$. Then we obtain four subsets of predicates: $P_{11} = \{p_2, p_5, p_7\}$, $P_2 = \{p_3, p_6, p_7\}$, $P_{31} = \{p_4, p_5, p_7\}$ and $P_{32} = \{p_4, p_6, p_7\}$, each defining a horizontal class fragment.
Finally, generating the horizontal fragments:
$Project_1$ given by clause cl_1 : (Duration ≤ 4) $\wedge$ (Cost() > 7000) $\wedge$ (Location = "Hong Kong").
$Project_2$ given by clause cl_2 : (Duration $= 2$) $\wedge$ (Cost() ≤ 7000) $\wedge$ (Location ="Hong Kong").
$Project_3$ given by clause cl_3 : ($5 \leq$ Duration ≤ 6) $\wedge$ (Cost() > 7000) $\wedge$ (Location = "Hong Kong").
$Project_4$ given by clause cl_4 : ($5 \leq$ Duration ≤ 6) $\wedge$ (Cost() ≤ 7000) $\wedge$ (Location = "Hong Kong").
$Project_5$ given by clause cl_5 : ELSE.

Complexity of Primary Algorithm The complexity of the primary algorithm is from steps 2 and 5, and that is $O(l*n^2)$ and $O(\alpha*(r+m))$, respectively. We note that n, l, α, r, and m represent the number of predicates, number of queries, number of fragments, number of attributes used by the queries, and the number of methods used by queries, respectively. Therefore, the complexity of our algorithm is: $O[l * n^2 + \alpha * (r + m)]$.

4 Derived Horizontal Class Fragmentation Algorithm

If the set of queries Q defined on C_i contains one or more component queries, then there should be some other class $C_j (j \neq i)$ in the class composition hierarchy which[3] is horizontally fragmented by the primary algorithm. Therefore, we can also horizontally fragment the class C_i based on the horizontal fragmentation of the class C_j. Such a fragmentation is known as *derived horizontal class fragmentation(DHCF)* [9].

4.1 Implicit object join

In order to support query predicates on complex attributes, object-oriented query languages usually use path expressions in which each attribute of a class C_i whose domain is a class C_j specifies an implicit join between C_i and C_j. Now we extend the definition of selection operation used in the relational model [7] to that on a path.

[3] There is a path from the class C_i to the class C_j with length $g (g \geq 1)$

Definition 5. (Selection operation) Let $P = C_1.A_1.A_2...A_n$ be a path; a selection operation is defined as : $\sigma_{<selectioncondition>}$ $(< C_1 >)$, where the symbol σ is used to denote the selection operator, and the selection condition is a qualification clause (Definition 1) specified on the attributes and methods of the class C_n in the path P. The class resulting from the selection operation has the same attributes and methods as C_1 which satisfies a condition defined on attributes and methods of C_n.

Definition 6. Let two classes C_i and C_j in a path P (C_j is the domain of an attribute a_k of C_i). We define *fan-out*(C_i, a_k, C_j) as the average number of C_j objects referred to by an object of C_i through attribute a_k. Similarly, the sharing level, *share*(C_i, a_l, C_j), is the average number of C_i objects that refer to the same object of C_j through attribute a_l [4]. We define *FAN*(C_i, C_j) and *SHARE*(C_i, C_j) as the average of *fan-out*(C_i, a_k, C_j) and average of *share*(C_i, a_l, C_j), over all the objects of class C_j and class C_i, respectively.

4.2 Algorithm

The inputs of the DHCF algorithm are the class C_i to be fragmented, and we assume that class C_j in the class hierarchy has been horizontally fragmented by PA into α fragments $\{F_1, F_2, ..., F_\alpha\}$. Note that each fragment $F_k (1 \leq k \leq \alpha)$ is defined by a qualification clause qc_k. The steps of the DHCF algorithm are :
1) For each clause $qc_k (1 \leq k \leq \alpha)$ determine its predicates $\{p_{k1}, p_{k2}, ..., p_{k|qc_k|}\}$. We recall that the $|qc_k|$ represents the cardinality of the qualification clause qc_k.
2) For each predicate $p_{kl} (1 \leq l \leq |qc_k|)$, determine the name of its attribute or method denoted by A_{kl}.

We can define the fragments defined by the DHCF algorithm as follow:
$f_k = \sigma_{\phi_{m=1}^{|qc_k|} < p_{km} >}(C_i)$ where ϕ is a logical connectives ($\wedge$, $\vee$) used in clause qc_k.

Example 3. We assume that the *Employee* class will be derived fragmented based on the horizontal fragments of the *Project* class. Let $Project_i (1 \leq i \leq 5)$ and $Emp_j (1 \leq j \leq 5)$ be respectively the project and employee class fragments. $Project_1$: (Duration ≤ 4) $\wedge$ (Cost() > 7000) $\wedge$ (Location = "Hong Kong"). In $Project_1$ there are three predicates $p_{11} : Duration \leq 4$, $p_{12} : Cost() > 7000$ and $p_{13} : Location =$ "*Hong Kong*", then Emp_1 is defined as:

$$Emp_1 = \sigma_{(E_1 \wedge E_2 \wedge E_3)} \text{ (Employee), where:}$$

$E_1 = (Employee.Dept.Proj.Duration \leq 4), E_2 = (Employee.Dept.Proj.Cost()$ $> \$7000)$ and $E_3 = (Employee.Dept.Proj.Location =$ "*Hong Kong*").
With the same technique we can determine others horizontal class fragments of Employee, i.e., Emp_2, Emp_3, Emp_4, Emp_5.

4.3 Correctness of the Derived Horizontal Fragmentation Algorithm

The fragments defined by primary horizontal fragmentation algorithm satisfy the correctness rules of completeness, reconstruction and disjointness [14].

However, the fragments defined by the derived horizontal fragmentation can overlap and thus do not satisfy the disjointedness property. We now present an algorithm called non_overlap described in figure 6. The inputs for this algorithm is the class C_i which is derived horizontal fragmented based on the horizontal fragmentation of class C_j. In order to know the overlapping instances of the class C_i, we check whether the fan-out(C_i, C_j) for each object instance of class C_i could be greater than 1. To do that, for each instance, we look in the class composition schema for the type of its complex attributes; if all these complex attributes are single valued, then fan-out(C_i, C_j) is 1. Otherwise, even if there is one complex attribute which is multi-valued the fan-out(C_i, C_j) could be greater than 1. Note that rigorous checking of whether two derived horizontal fragments are overlapping or not would require examining each object instance of class C_i. In case there is a multi-valued object-based instance variable, we execute the non_overlap algorithm. We note that this algorithm ensures the disjointedness property. In order to facilitate the maintenance of the database, we create a table with three columns: the first column contains the OID(s) of the instance(s) eliminated by non_overlap algorithm, the second contains the fragment which includes these instances, and the third column contains the name of the fragment(s) from which these instances were eliminated. We call this table overlap catalog. The overlap catalog facilitates the identification and management of objects which belong to more than one horizontal fragment.

begin
  For each *overlapping* instance I_j do
    For each fragment that contains I_j do
      Compute the queries access frequency to I_j
      Take the fragment with *maximal access frequency*, which is called F_i
      For each fragment F_l ($l \neq i$) do $F_l = F_l - I_j$
end

Fig. 6. The non_overlap algorithm

5 Conclusion

In this paper, we have studied the role of queries in the horizontal fragmentation in the object-oriented model, which has not been addressed adequately by researchers. We have proposed two horizontal fragmentation algorithms: primary algorithm and derived algorithm. We have established the complexity of the *primary horizontal fragmentation* which is $O(l * n^2)$, (n and l represent the number of queries and their predicates, respectively) and there better than earlier work. The fragments defined by the *derived horizontal algorithm* can overlap, then we have proposed a method in order to eliminate overlapping instances. We are now evaluating the performance of our algorithm and studying the impact of the update queries defined on the resulting fragments.

References

1. M. Atkinson, F. Bancilhon, F. DeWitt, K. Dettrich, D. Maier, and S. Zdonik. The object database system manifesto. *in Proceeding of the first International Conference on Deductive, Object-Oriented Databases*, pages 40–57, 1989.
2. J. Banerjee, K Kim, and K. C. Kim. Queries in object oriented databases. *in Proceedings of the IEEE Data Engineering Conference*, February 1988.
3. L. Bellatreche, A. Simonet, and M. Simonet. An algorithm for vertical fragmentation in distributed object database systems with complex attributes and methods. *in International Workshop on Database and Expert Systems Applications (DEXA'96), Zurich*, pages 15–21, September 1996.
4. E. Bertino and C. Guglielmina. Path-index: An approach to the efficient execution of object-oriented queries. *Data & Knowledge Engineering*, 10:1–27, 1993.
5. E. Bertino, M. Negri, G. Pelagatti, and L. Sbattella. Object-oriented query languages: The notion and the issues. *IEEE Transactions on Knowledge and Data Engineering*, 4(3):223–237, 1992.
6. S. Ceri, M. Negri, and G. Pelagatti. Horizontal data partitioning in database design. *Proceedings of the ACM SIGMOD International Conference on Management of Data. SIGPLAN Notices*, 1982.
7. R. ElMasri and S. B. Navathe. *Fundamentals of Database Systems.* Benjamin Cummings, Redwood City, CA, 1994.
8. C. I. Ezeife and K. Barker. A comprehensive approach to horizontal class fragmentation in distributed object based system. *International Journal of Distributed and Parallel Databases*, 1, 1995.
9. K. Karlapalem and Q. Li. Partitioning schemes for object oriented databases. *in Proceeding of the Fifth International Workshop on Research Issues in Data Engineering- Distributed Object Management, RIDE-DOM'95*, pages 42–49, March 1995.
10. K. Karlapalem, Q. Li, and S. Vieweg. Method induced partitioning schemes in object-oriented databases. *in 16th International Conference on Distributed Computing System (ICDCS'96), Hong Kong*, May 1996.
11. K. Karlapalem, S.B. Navathe, and M. M. A. Morsi. Issues in distributed design of object-oriented databases. In *Distributed Object Management*, pages 148–165. Morgan Kaufman Publishers Inc., 1994.
12. S. J. Lim and Y. K. Ng. A formal approach for horizontal fragmentation in distributed deductive database design. *in the 7th International Conferences on Database and Expert Systems Applications (DEXA'96), Lecture Notes in Computer Science 1134, Zurich*, pages 234–243, September 1996.
13. S.B. Navathe, S. Ceri, G. Wiederhold, and Dou J. Vertical partitioning algorithms for database design. *ACM Transaction on Database Systems*, 9(4):681–710, December 1984.
14. S.B. Navathe, K. Karlapalem, and M. Ra. A mixed partitioning methodology for distributed database design. *Journal of Computer and Software Engineering*, 3(4):395–426, 1995.
15. S.B. Navathe and M. Ra. Vertical partitioning for database design : a graphical algorithm. *ACM SIGMOD*, pages 440–450, 1989.
16. M. T. Özsu and P. Valduriez. *Principles of Distributed Database Systems.* Prentice Hall, 1991.

This article was processed using the LaTeX macro package with LLNCS style