Query Optimization using Horizontal Class Partitioning in Object Oriented Databases

Ladjel Bellatreche **Kamalakar Karlapalem** **Ana Simonet**

Department of Computer Science
University of Science & Technology
Clear Water Bay Kowloon
Hong Kong
{ladjel, kamal}@cs.ust.hk

Laboratoire TIMC-IMAG
Faculté de Medecine
La Tronche 38700
France
Ana.Simonet@imag.fr

Résumé

Le traitement des requêtes reste un des enjeux les plus importants dans les bases de données orientées-objet (BDDOOs). Pour effectuer l'optimisation physique des requêtes, il est nécessaire de disposer d'un modèle de coût pour leur évaluation. Le fragmentation horizontale de classes (FHC) est une technique qui permet de réduire le nombre d'accès au disque afin d'exécuter un ensemble donné de requêtes en minimisant le nombre d'accès aux objets. L'utilité de la FHC pour la minimisation du temps d'exécution pour les requêtes est largement reconnu. Toutefois, la plupart des modèles de coût existants pour les BDDOOs ne prennent pas en considération le critère de FHC. Dans ce papier, on présente un modèle analytique de coût pour l'exécution des requêtes pour des classes non partitionnées et des classes horizontalement partitionnées. Un tel modèle est nécessaire pour décider si on fragmente les classes d'une BDDOO ou non. Nous analysons l'effet de la FHC sur l'évaluation des requêtes.
Mots-clés : Base de données orientées objet, fragmentation horizontale, modèle de coût, traitement de requête, optimisation de requête, évaluation de performance.

Abstract

Query processing remains one of the most important challenges of object oriented database systems (OODBSs). A cost model for query processing is very useful in performing the physical query optimization. Horizontal class partitioning (HCP) is a technique for reducing the number of disk accesses for executing a given set of queries by minimizing the number of irrelevant object instances accessed. Moreover, its importance in reduction of query execution time has been widely acknowledged. However, existing cost models for query processing in OODBSs do not take into consideration the HCP criteria. In this paper, we present an analytical cost model for query execution for unpartitioned and horizontally partitioned classes. This cost is necessary for deciding whether to use HCP or to keep all classes unpartitioned. The effect of HCP upon the query execution process is analyzed.
Key Words : Object-oriented database, horizontal class partitioning, cost model, query processing, query optimization, performance evaluation.

1 Introduction

Query processing is one of the crucial issues in object oriented database systems (OODBSs), and an important issue related to query processing concerns optimization techniques and access structures for reducing query processing costs [3]. Horizontal partitioning is often used as a means to achieve better performance of database systems by reducing the disk accesses required to execute a query by minimizing the number of irrelevant objects accessed [15]. In centralized databases, data is stored together with all the attributes that define instances. When this data is retrieved by some query, all the instances are loaded into the main memory, even though only some of them are needed. If the number of retrieved pages is high, the number of irrelevant instances accessed over the relevant instances can also get high. If we can have only those instances needed by the query loaded into the main memory, then the number of retrieved pages will be smaller. Therefore, if a class is partitioned into class fragments of instances in which the more frequently accessed instances for the queries are placed in the same fragment, the number of pages accessed can be reduced [15]. Such a partitioning is known as *horizontal class partitioning (HCP)*. The HCP is known to reduce the amount of irrelevant data accessed by the queries in both relational databases [7], [22], and [20], and object oriented databases [2] and [9]. Thus, a partial list of benefits of HCP includes: 1) Different queries access or update only the needed fragments of a class, so partitioning will reduce the amount of irrelevant data accessed by the queries, 2) Partitioning reduces the amount of data transferred when processing of distributed queries is required [9], 3) The decomposition of a class into horizontal class fragments (HCFs), each being treated as a unit, permits queries to be executed concurrently, 4) Further, HCP can be treated as an object oriented database schema design technique [14], which takes a class and generates a class hierarchy of HCFs so as to increase the efficiency of query execution, while providing additional semantics for the object oriented database scheme.

Research has been done on cost function modeling [3, 5, 12, 21]. All these techniques assume that all classes in OODBs are unpartitioned, except the work in [10] which proposed an analytical cost model for a vertically partitioned class taking into account input/output costs. However, the HCP algorithms either in relational or object oriented models ignored the physical costs corresponding to the savings in the amount of irrelevant objects accessed. The utility of the HCP can be measured by the amount of savings in disk accesses for query execution. According to Karlapalem et al. [15], two factors, I/O operations and data transfer, are the most important factors for the performance of the applications in distributed database systems.

1.1 Related Work

Ezeife et al. [9] presented a set of algorithms for horizontally fragmenting based on 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 an algorithm developed by [22] which generates minterm predicates [7] from simple predicates defined in the queries. Bellatreche et al. [2] presented two algorithms for generating HCFs of a given class: primary and derived horizontal class partitioning algorithms. The primary algorithm is an adaptation of the vertical partitioning algorithm proposed in the relational model [20]. The approach used is based on affinity between predicates. They finalized derived horizontal partitioning concept and developed a scheme for generating derived HCFs. But the work addressed in [2, 9] ignored the physical cost corresponding to the savings in the amount of irrelevant data accessed. In [16] presented a cost model for executing queries on both *vertically* partitioned class collection and unpartitioned class collection. But there is no cost model for query execution taking into account HCF.

In this paper, we present an analytical cost model for unpartitioned classes and horizontally partitioned classes based on a given set of queries in OOBDSs, and we show the utility of HCP in OODBSs. The main contributions of this paper are:
1. Development of a cost model for executing queries on both horizontally partitioned class collection and unpartitioned class collection.
2. Show the utility of HCP.
3. Evaluation of utility of HCP based on fan-out across class composition hierarchy, and cardinality of classes.

The rest of the paper is organized as follows : section 2 presents the data model and some definitions used in specifying the horizontal class partitioning in an OODBS, section 3 presents the cost model for unpartitioned classes and horizontally partitioned classes, in section 4, the utility of HCP in query processing is evaluated, and section 5 presents a summary and mentions further work that is currently being undertaken.

2 HCP in Object Oriented Databases

2.1 Data Model

The model we have retained is built around the fundamental concept of an object. An object represents a real entity and it is an abstraction defined by *(a)* an unique object identifier (OID) which remains invariable during the life of the object [1],*(b)* a set of attribute values which define the state of the object, and *(c)* an interface consisting of methods which manipulate the object to form an encapsulated object. Objects having the same attributes and methods are grouped into a class. An instance of a class is an object with an OID and 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. A class in an object oriented database is represented by a set of attributes A and a set of methods M. For each attribute, the set of values it may have is defined by its domain. Two types of attributes are possible (simple and complex) [1]: a simple attribute can only have an atomic domain (e.g., integer, string, etc.), a complex attribute has a database class as its domain. Thus, there is a hierarchy which arises from the aggregation relationship between

[1]The OID is used as a means of referencing the object by other object(s)

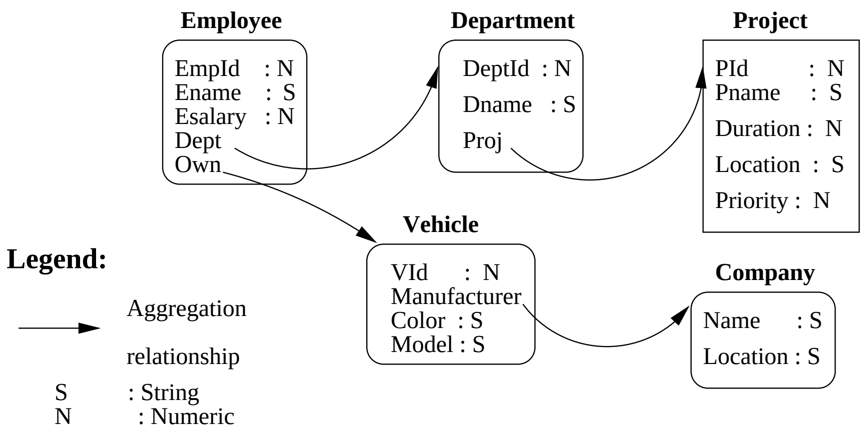

Figure 1: The Class Composition Hierarchy of the class Employee

the classes and their attributes. This hierarchy is known as class composition hierarchy [17] which is a rooted directed graph (RDG), with the nodes as the classes, and an arc between a pair of classes C_1 and C_2, if C_2 is the domain of an attribute of C_1. We note that the classes with only simple attributes represent the leaves of RDG. Furthermore, attributes can be single valued or multi-valued. The multi-valued attributes are defined by using constructor denoted as *Const* such as *set*, *list*, *tree*, and *array* [6]. 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). Karlapalem el al. [13] identified two types of methods: simple and complex. A method which does not call/invoke any other method is called a simple method, otherwise it is complex method.

2.2 Basic Concepts

Before describing the HCP in OODBSs, some definitions are presented.

Definition 1 (Simple predicate [7]) *A simple predicate is a predicate defined on a simple attribute or on a simple method and it is defined as :* attribute/method operator value*, where operator is a comparison operator* $(=, <, \leq, >, \geq, \neq)$ *. The* value *is chosen from the domain of the attribute or the value returned by the method.*

Definition 2 *A path P [5] represents a branch in a class composition hierarchy and it is specified by:*
$P : C_1.A_1.A_2....A_n$ *(*$n \geq 1$*) where :* C_1 *is a class in the database schema,* A_1 *is an attribute of class* C_1*, and* 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* m_n *of this class which returns a set of values or set of OIDs. The length of the path* P *is defined by the number of attributes,* n*, in* P*. We call the first class* C_1 *the* starting class *and the last attribute (or method)* A_n *the* ending attribute or method *of the path [8].*

We consider a typical object query language which is of form:

SELECT "result list"
FROM "target class"
WHERE "qualification clause"

The "result list" involves only attributes or methods from a single class. The "quantification clause " defines a boolean combination of predicates by using the logical connectives: $\wedge$, $\vee$, $\neg$. A predicate can be defined on path expression. In this case, we call this predicate *component predicate* [2]. The queries that we consider for the purpose paper are single-target queries [4]. A single-target query retrieves attributes or methods from only the "target class". Other classes, however, may be used in the query based on their relationships with the "target class".

Example 1 *The following query returns the name of all employees working in a project located at "Hong Kong":*

SELECT Ename
FROM Employee
WHERE Employee.Dept.Proj.Location = "Hong Kong"

In this query, the starting and ending class of the path is Employee and Project classes, respectively.

2.3 Predicate Affinity Algorithm

The HCFs of a class C which we will use in this paper are obtained by using the predicate affinity algorithm introduced in [2]. This algorithm has a set of queries and their access frequencies as input which respect the 80/20 rule, that is 20% of user queries account for 80% of the total data access in database systems [19]. It uses the affinity between predicates. Predicates having a high affinity are grouped together to form a HCF. The algorithm starts by performing an analysis on the predicates from a set of queries defined on the class C. From these queries, we obtain the *use* parameter as: $use(q_h, p_l) = 1$ means that the predicate p_l is used by the query q_h, 0 otherwise. The *use* values are used to generate a *predicate usage matrix* and then a *predicate affinity matrix*, where each value $(p_l, p_{l'})$ represents the sum of the frequencies of queries which access predicates p_l and $p_{l'}$ simultaneously. After that, we apply the graph-based algorithm [20] in order to group these predicates into sets of predicates. In each group, we optimize (if it is possible) by using some implications between the predicates defined in queries. After that, we generate the HCFs. Each HCF is defined by a boolean combination of predicates using the logical connectives ($\wedge$, $\vee$).

We note that the reconstruction of a global class C from its HCFs $\{F_1, F_2, ..., F_m\}$ is performed by the union operator of all HCFs. Thus, $C = \bigcup_{i=1}^{m} F_i$.

2.4 Internal Representation of HCFs of a Class

Let C be a class to be horizontally partitioned into m HCFs $F = \{F_1, F_2, ..., F_m\}$, where each HCF F_i must satisfy qualification clause. A horizontal partitioning of a class C can be

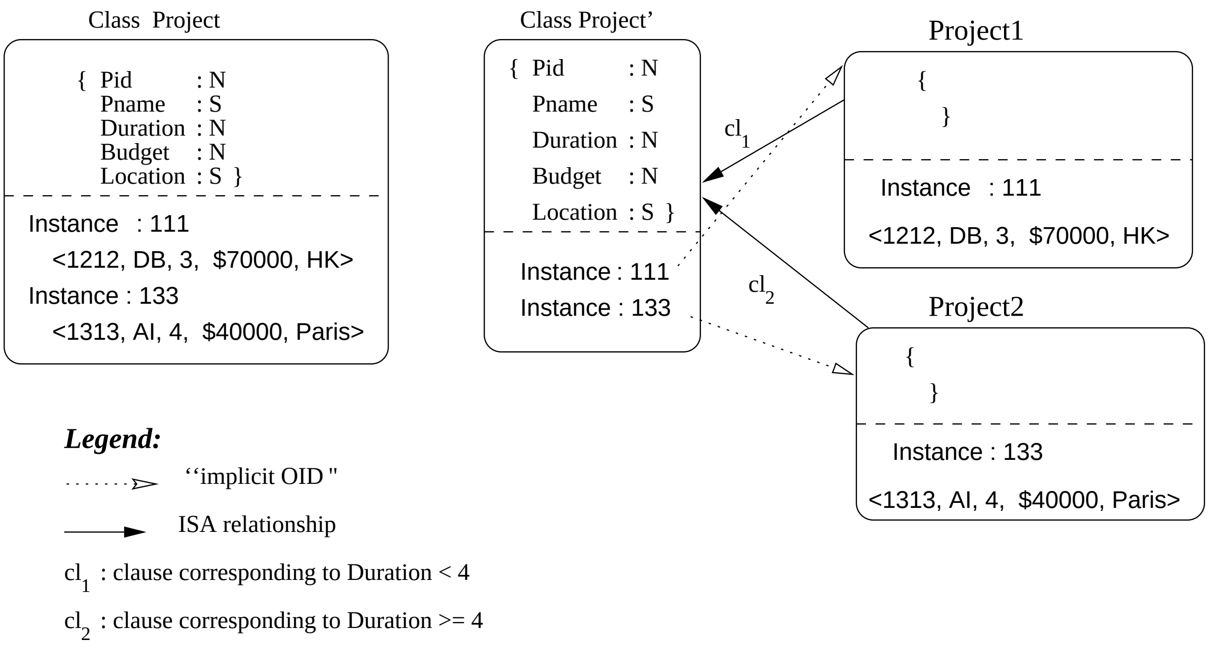

Figure 2: Horizontal class fragments (Project1, Project2) of the Class Project

represented as a *isa hierarchy* [15], where each HCF is specified as a class. Let assume that the *Project* class has been horizontally partitioned into two HCFs *Project1* and *Project2* representing all projects with duration less than 4, and all projects with duration greater than or equal to 4, respectively. Figure 2 illustrates the internal representation of an non-overlapping HCFs the class *Project*. A class *Project'* is created with the same of attributes as class *Project*. This class *Project'* has no explicit attributes, and all its instances are implicit.

That is, there is no object that has the class *Project'* as its base class. For each HCF a class is created with the same set of attributes as class *Project*. Each HCF will have only explicit instances, and these instances will be those that satisfy the qualification clause. This representation helps us to define our cost model for executing query on horizontally partitioned classes.

2.5 The Effect of HCP on Query Execution

An important requirement of an object oriented query language is the capability of navigating through the object structures [6]. This operation is performed over objects along a path. Path is the most widely used, but it is a costly operation for exploring logical relationships among complex objects in queries. There is a lot of work done to optimize object-oriented queries involving path expressions. This work involved the definition of new object algebra [23], complex object assembly [18], and cost model [3].

We present an example to give an overview on how HCP technique can reduce the cost of query execution.

Example 2 *Let us consider the following query which consists of finding out all the em-*

ployees in Figure 1 who are working on projects with duration $= 3$. *The qualification clause of this query is the component predicate p defined as p: e.Dept.Proj.Duration* $= 3$, *where e is a variable denoting an object in the class Employee. The predicate p is performed from the objects in class Employee to the objects in the class Project via the attribute Dept, the class Department, the attribute Proj and the class Project. Let class Project in Figure 1 be horizontally partitioned into two HCFs: Project1 and Project2 as shown in Figure 2. In this case, the component predicate p will be refined into two component predicates:* p_1 *and* p_2*, where* p_1 *is performed from the objects in class Employee to the objects in the HCF Project1 via the attribute Dept, the class Department, the attribute Proj and the class Project1, and* p_2 *is performed from the objects in class Employee to the objects in the HCF Project2 via the attribute Dept, the class Department, the attribute Proj and the class Project2. Then, our query accesses only the objects defined by the predicate* p_1 *(i.e., Project1), and it does not need to access Project2, because Project2 does not contribute to the answer of the query.*

3 A Cost Model for Executing a Query

Assume a database of object instances stored on secondary memory which is divided into pages of fixed size. In this section, we present two analytical cost models of executing a set of k queries $\{q_1, q_2, ..., q_k\}$, where every query may contain a simple or component predicates. The first cost is for executing these queries on unpartitioned classes (i.e., all classes in class composition hierarchy are unpartitioned), and second one is for executing the same queries on horizontally partitioned class(es) (i.e, we assume that there are some class(es) which are horizontally partitioned using predicate affinity algorithm described in section 2.3. The objective of our cost models is to calculate the cost of executing these queries, each of which accesses a set of objects. The cost of a query is directly proportional to the number of pages it accesses. Costs are estimated in terms of disk page accesses.

The total cost to execute a query is given by [10]:

$$\textbf{Total\_Cost} = \textbf{IO\_Cost} + \textbf{CPU\_Cost} + \textbf{COM\_Cost}$$

where *IO_Cost* is the input/output cost for reading and writing data between memory and disk, CPU_Cost is the cost of executing CPU instructions, (for example, for evaluating predicate), and COM_Cost is the cost of network communication among different nodes. In this paper, as in [10], we concentrate on the *IO_Cost* and disregard the *CPU_Cost* and COM_Cost. This is because for large database applications with huge number of data accesses, the *CPU_Cost* contribution towards the Total_Cost will not be significant [10], and the database we consider is centralized, therefore the COM_Cost is not taken into account.

3.1 Cost Model Parameters

Definition 3 (Fan-out and Share[5]) *Let two classes* C_i *and* C_j *be 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*

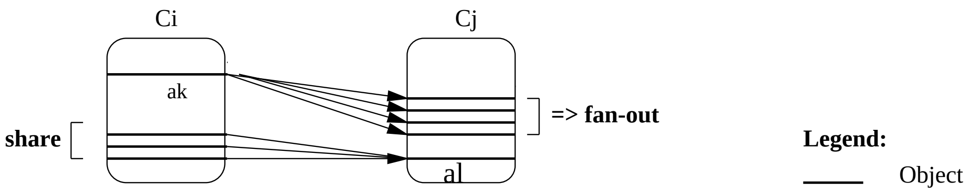

Figure 3: Fan-out$(C_i,\ a_k,\ C_j)$

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*. 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.*

We itemize the input parameters used in developing the cost model for unpartitioned and horizontally partitioned classes. We classify the parameters for the cost model into three categories: database, query and horizontal class fragments.

a- Database Parameters:

- $||C_i||$: cardinality of a class collection C_i (i.e., number of objects)
- $|C_i|$: total number of pages occupied by the class C_i
- LC_i : object length/size (in bytes) in the class collection C_i
- PS : page size of the file system (in bytes)

b- Query Parameters :

- C_s^h : starting class of a component predicate p_l
- REF_i: the number of objects references for class C_i during the path evaluation process along the class composition hierarchy
- k : number of queries
- SEL_h : selectivity of query q_h
- $FAN(C_{i-1}, C_i)$: the fan-out for the class composition hierarchy from class C_{i-1} to C_i
- L_{proj} : length of output result

c- Horizontal class fragments Parameters :

- m_i : number of HCFs of class C_i
- $||F_j||$: cardinality of a HCF F_j (i.e., number of objects)
- $|F_j|$: total number of pages occupied by the HCF F_j
- ref_j: the number of objects references for HCF F_j during the path evaluation process along the class composition hierarchy
- $valid(p_l, F_j)$: binary variable, it is of value 1 if the predicate p_l is valid in HCF F_j, 0 otherwise.

Values for inputs parameters like ($||C_i||$, LC_i, PS, k, FAN, L_{proj}) may be provided in several ways [3], as in the case of relational databases. They can provided by the database administrator, or can be automatically acquired by the system through data inspection or sampling. The selectivity can be estimated by using the techniques described in [3]. The HCFs m_i of a class C_i is obtained by applying the predicate affinity algorithm [2]. The cardinality of every HCF F_j is calculated as : $||F_j|| = sel(F_j) \times ||C_i||$ for all $1 \leq j \leq m_i$, where $sel(F_j)$ represents the selectivity of HCF F_j.

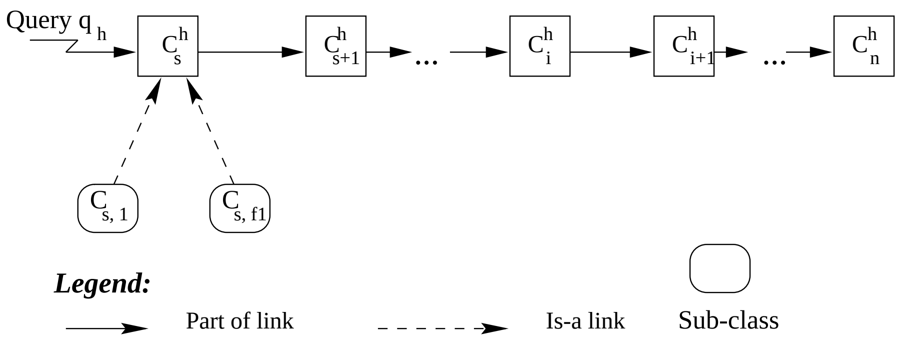

Figure 4: Example of Unpartitioned Classes

3.2 Cost of Executing a Single Query on Unpartitioned Classes

In order to make the formulation of the I/O cost for both unpartitioned and horizontally partitioned classes easier, we define it for one query q_h as shown Figure 4, we assume that q_h contains one component predicate p_l. In the next subsection, we shall generalize this formulation for all k queries. We assume that the starting class of component predicate as C_s^h and ending class as C_n^h. The length of path from C_s^h to C_n^h is n.

Our cost model is based to [11, 12]'s formulation. The cost can be broken up into 2 components: the cost of evaluating the predicate, and the cost of building the output result which corresponds to the number of pages to be loaded in order to evaluate a predicate, and the number of pages to be loaded to build the result, respectively.

3.2.1 Estimation of the Number of Pages in a Class Collection

Sequential scan and index scan are two major strategies used for scanning a class collection [10]. The objective of using an index is to attain faster instances access, while the objective of using HCP is to reduce irrelevant instances access. Then the two objectives are orthogonal and complementary. We concentrate on the sequential scan strategy [10]; the use of index can also be incorporated into our model naturally as needed. The objects of a class are assumed to be stored/accessed on/from the disk sequentially. We assume that the objects are smaller than the page size and not to cross page boundaries. The total number of pages occupied by a class collection C is given by:

$$|C| = \left\lceil \frac{||C||}{\lfloor \frac{PS}{LC} \rfloor} \right\rceil \tag{1}$$

where $\lceil\ \rceil$ and $\lfloor\ \rfloor$ are the ceiling and floor functions respectively. The formula can be used for a HCF F as follows:

$$|F| = \left\lceil \frac{||F||}{\lfloor \frac{PS}{LC} \rfloor} \right\rceil \tag{2}$$

3.2.2 Estimation of the Number of Page Accesses for Predicate Evaluation

Let q_h be a query having a component predicate p_l. We first define the number of distinct references involved in the path [12] $REF_i = (1 - Prob_i) \times ||C_i^h||$, where $Prob_i$ is the probability of an object of collection C_i^h to be not involved in the path, we have:

$$Prob_i = (1 - \frac{1}{||C_i^h||})^{(REF_{i-1} \times sel_l \times FAN(C_{i-1}^h, C_i^h))}, \text{ with } REF_1 = ||C_s^h||.$$

In order to estimate the number of page accesses in class collection C_i^h (see Figure 4) during the evaluation of a predicate, we use the Yao function [24]: Given n records uniformly distributed into m blocks ($1 \leq m \leq n$), each contains n/m records. If k records ($k \leq n$) are randomly selected from n records, the expected number of page access is given by:

$$\mathsf{Yao}(\mathsf{n}, \mathsf{m}, \mathsf{k}) = \mathsf{m} \times \left[1 - \prod_{\mathsf{i}=1}^{\mathsf{k}} \frac{\mathsf{n} \times \mathsf{d} - \mathsf{i} + 1}{\mathsf{n} - \mathsf{i} + 1} \right] \tag{3}$$

where $d = 1 - \frac{1}{m}$.
We use the Yao's formula with $\mathsf{n} = ||\mathsf{C}_\mathsf{i}^\mathsf{h}||$, $\mathsf{m} = |\mathsf{C}_\mathsf{i}^\mathsf{h}|$ and $\mathsf{k} = \mathsf{REF_i}$ is defined above.

3.2.3 Cost Formulate for Query Execution

The total cost for executing the query q_h having a component predicate p_l defined on unpartitioned classes is given by the following equation:

$$\mathsf{Total\_Cost} = \mathsf{Predicate\_Evaluation\_Cost} + \mathsf{IO\_Building\_Output} \tag{4}$$

with:

$$\mathsf{Predicate\_Evaluation\_Cost} = \sum_{\mathsf{i}=\mathsf{s}}^{\mathsf{n}} \mathsf{Yao}(||\mathsf{C}_\mathsf{i}^\mathsf{h}||, |\mathsf{C}_\mathsf{i}^\mathsf{h}|, \mathsf{REF_i}) \tag{5}$$

$$\mathsf{IO\_Building\_Output} = \left\lceil \frac{\mathsf{SEL_h} \times ||\mathsf{C}_\mathsf{s}^\mathsf{h}||}{\lfloor \frac{\mathsf{PS}}{\mathsf{Lproj}} \rfloor} \right\rceil \tag{6}$$

3.2.4 Cost of Executing a Set of Queries on Unpartitioned classes

We now, generalize the total cost for executing all k queries $\{q_1, q_2, ..., q_k\}$ which is given by the following equation:

$$\mathsf{Total\_Cost} = \mathsf{Predicate\_Evaluation\_Cost} + \mathsf{IO\_Building\_Output} \tag{7}$$

where:

$$\mathsf{Predicate\_Evaluation\_Cost} = \sum_{\mathsf{h}=1}^{\mathsf{k}} \left[\sum_{\mathsf{i}=\mathsf{s_h}}^{\mathsf{n}} \mathsf{Yao}(||\mathsf{C}_\mathsf{i}^\mathsf{h}||, |\mathsf{C}_\mathsf{i}^\mathsf{h}|, \mathsf{REF_i}) \right] \tag{8}$$

$$\mathsf{IO\_Building\_Output} = \sum_{\mathsf{h}=1}^{\mathsf{k}} \left[\left\lceil \frac{\mathsf{SEL_h} \times ||\mathsf{C}_\mathsf{s}^\mathsf{h}||}{\lfloor \frac{\mathsf{PS}}{\mathsf{Lproj}} \rfloor} \right\rceil \right] \tag{9}$$

We note that all these equations are for queries having one component predicates. For a query having more than one predicate, we calculate the I/O cost for *each predicate* based on above equations, and finally we sum up all these I/O costs.

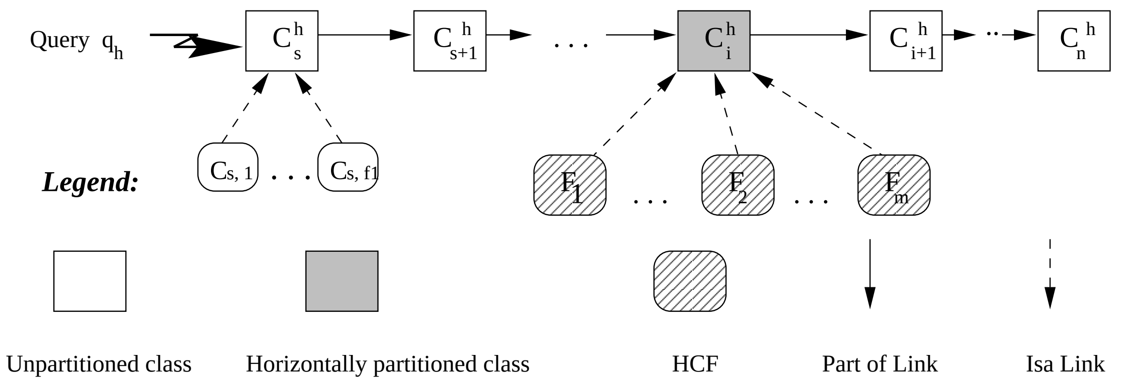

Figure 5: Example Horizontal Class Partitioning

3.3 Cost of Executing a Single Query on Horizontally Partitioned Class

In this subsection, we assume there is one class C^h_i in the class composition hierarchy be horizontally partitioned into m_i HCFs. Let us consider a query q_h having a component predicate invoking C^h_i as shown in Figure 5. Before describing the cost on horizontally partitioned classes, we first estimate the number of page accesses in HCF F_j during the evaluation of a predicate. We can apply the same formula for unpartitioned class (see subsection 3.2.2) and that is due to the representation of HCF as a class. Then the number of page accesses is given by: $\mathsf{Yao}(||\mathsf{F_j}||, |\mathsf{F_j}|, \mathsf{ref_j})$ where: $\mathsf{ref_j} = (1 - \mathsf{Prob_j}) \times ||\mathsf{F_j}||$ where $Prob_j$ is the probability of an object of HCF F_j to be not involved in the path, we have:

$$Prob_i = (1 - \frac{1}{||F_j||})^{(REF_{i-1} \times sel_l \times FAN(F_j, C^h_i))}$$

The predicate evaluation cost (IO_Eval_Cost) is the sum of Yao's function over the class involved by the query, but we need to consider the effect of horizontal partitioning of class C^h_i which is represented by the binary variable $valid(p_l, F_j)$ having the value 1 if the predicate p_l is valid in HCF F_j, 0 otherwise. That means we load only the HCFs which satisfy the predicates defined by the query.

$$\mathsf{IO\_Eval\_Cost} = \sum_{(i=s, i \neq h)}^{n} \mathsf{Yao}(||\mathsf{C^h_i}||, |\mathsf{C^h_i}|, \mathsf{REF_i}) + \sum_{i=1}^{m_i} \mathsf{valid}(\mathsf{p_l}, \mathsf{F_i}) \times \mathsf{Yao}(||\mathsf{F_i}||, |\mathsf{F_i}|, \mathsf{ref_i}) \quad (10)$$

and the cost of building the output result is:

$$\mathsf{IO\_Building\_Output} = \left\lceil \frac{\mathsf{SEL} \times ||\mathsf{C^h_s}||}{\lfloor \frac{\mathsf{PS}}{\mathsf{Lproj}} \rfloor} \right\rceil \quad (11)$$

We generalize the above formulas for i ($i > 1$) classes which are horizontally partitioned. We introduce a binary variable H_i as:

$$\mathsf{H_i} = \begin{cases} 1 & \text{if the class } \mathsf{C_i} \text{ is horizontally partitioned.} \\ 0 & \text{Otherwise.} \end{cases}$$

The predicate evaluation cost will be defined by the following equation:

$$\mathsf{IO\_Eval\_Cost} = \sum_{i=s}^{n} \left[\mathsf{H_i} \times \sum_{j=1}^{m_i} (\mathsf{valid}(\mathsf{p_l}, \mathsf{F_j}) \times \mathsf{Yao}(||\mathsf{F_j}||, |\mathsf{F_j}|, \mathsf{ref_j}) + (1 - \mathsf{H_i}) \times \mathsf{Yao}(||\mathsf{C_i^h}||, |\mathsf{C_i^h}|, \mathsf{REF_i}) \right] \tag{12}$$

The I/O building output result is given by:

$$\mathsf{IO\_Building\_Output} = (1 - \mathsf{H_s}) \times \left\lceil \frac{\mathsf{SEL} \times ||\mathsf{C_s^h}||}{\lfloor \frac{\mathsf{PS}}{\mathsf{Lproj}} \rfloor} \right\rceil + \mathsf{H_s} \times \sum_{j=1}^{m_s} (\mathsf{valid}(\mathsf{p_l}, \mathsf{F_j}) \times \left\lceil \frac{\mathsf{SEL} \times ||\mathsf{F_j}||}{\lfloor \frac{\mathsf{PS}}{\mathsf{Lproj}} \rfloor} \right\rceil \tag{13}$$

3.4 Comparison of Total Cost for Unpartitioned and Partitioned Classes

In this subsection, we compare the total cost formula for unpartitioned classes with the horizontally partitioned classes, in order to show the utility of HCP. In unpartitioned classes, all instances of these classes are loaded into main memory (see equation 5), but in horizontally partitioned classes; the query needs to load only the HCFs which contribute to the result, that it is done by the valid function (see equation 10). We note that $|F_j|$ is always less than $|C_i|$ for horizontally partitioned classes, and then $Yao(||F_j||, |F_j|, ref_j) < Yao(||C_i^h||, |C_i^h|, REF_i)$. In this case, the HCP will always improve the performance of evaluating the predicate. But if the selectivity of the query is very high, which means all HCFs are concerned by this query. In this case, there is extra overhead which is due to the cost of applying the *union operation* of all HCFs (see section 2.3) reconstruct the result of a query. This overhead may deteriorate the performance of the HCP.

4 Evaluation of the Utility of Horizontal Partitioning

4.1 Performance Metric

In order to show the utility of HCP, we conduct some analytical experiments to see the effect of fan-out, and the cardinality on the improvement of performance due to HCP. This improvement of performance is characterized by the normalized IO metric defined by [10] as follow:

$$\mathsf{Normalized\ IO} = \frac{\mathsf{\#\ Of\ IOs\ for\ the\ horizontally\ partitioned\ class\ collection}}{\mathsf{\#\ Of\ IOs\ for\ the\ unpartitioned\ class\ collection}}$$

We note that the value of normalized IO less than 1.0 implies HCP is beneficial. In order to present interesting results, while maintaining the control over the number of parameters and studying the impact of parameter changes, we consider the following parameters which are: cardinality of root class (Class Employee), page size (PS), number of objects per page (LC), number of HCFs per class, Fan-out of a class along the class composition hierarchy (Employee $\rightarrow$ Department $\rightarrow$ Project). We will study the following cases of horizontal partitioning:

- HP_1- only the class *Project* is horizontally partitioned.
- HP_2- both the classes *Project* and *Department* are horizontally partitioned.
- HP_3- all the classes *Employee*, *Department* and *Project* are horizontally partitioned.

The reason for selecting such a class collection is that it enables us to study both the impact of fan-out along the class composition hierarchy, and also the effect of horizontal partitioning classes along the class composition hierarchy. Let us assume that the classes *Employee*, *Department* and *Project* are horizontally partitioned as follows:

Horizontal Class Fragments of the Class Project
$Project_1$ given by clause cl_1 : (Duration $\leq$ 4) $\wedge$ (Cost( ) $\leq$ 7000) $\wedge$ (Location = "Hong Kong")
$Project_2$ given by clause cl_2 : (Duration $\leq$ 4) $\wedge$ (Cost( ) $>$ 7000) $\wedge$ (Location ="Hong Kong")
$Project_3$ given by clause cl_3 : (Duration$\leq$ 4) $\wedge$ (Cost() $\leq$ 7000) $\wedge$ (Location = "Paris")
$Project_4$ given by clause cl_4 : (Duration $\leq$ 4) $\wedge$ (Cost( ) $>$ 7000) $\wedge$ (Location = "Paris")
$Project_5$ given by clause cl_5 : Duration $>$ 4

Horizontal Class Fragments of the Class Department
$Depart_1$ given by clause cl_1 : Dname = "Computer Science"
$Depart_2$ given by clause cl_2 : Dname = "Mathematics"

Horizontal Class Fragments of the Class Employee
Emp_1 given by clause cl_1 : Esalary $\leq$ 13000
Emp_2 given by clause cl_2 : Esalary $>$ 13000

We consider 15 queries which will be used on our experiments and they are classified into 6 types as follows:

Type 1: Queries Accessing only Class Employee	**SEL**
Select Ename From Employee Where Salary $\leq$ 10000	0.05
Select EmpId, Ename From Employee Where Salary $>$ 28000	0.5
Select Salary From Employee Where Ename = "Dupond"	0.1

Type 2: Queries Accessing only Class Employee & Department	**SEL**
Select Ename From Employee Where Dept.Dname ="CS"	0.05
Select EmpId, Ename From Employee Where Salary $\leq$ 10000 And Dept.Dname ="Mathematics"	0.5

Type 3: Queries Accessing all 3 Classes	**SEL**
Select Ename From Employee Where Esalary $\leq$ 10000 And Dept.proj.Location = "Hong Kong"	0.05
Select EmpId, Ename From Employee Where Salary $\leq$ 28000 And Dept.Proj.Duration $>$ 4	0.5
Select Salary From Employee Where Dept.Proj.Location = "Paris"	0.1

Type 4: Queries Accessing only Class Department	**SEL**
Select DeptId From Department Where Dname = "Mathematics"	0.05
Select DeptId From Department Where Dname = "CS"	0.5

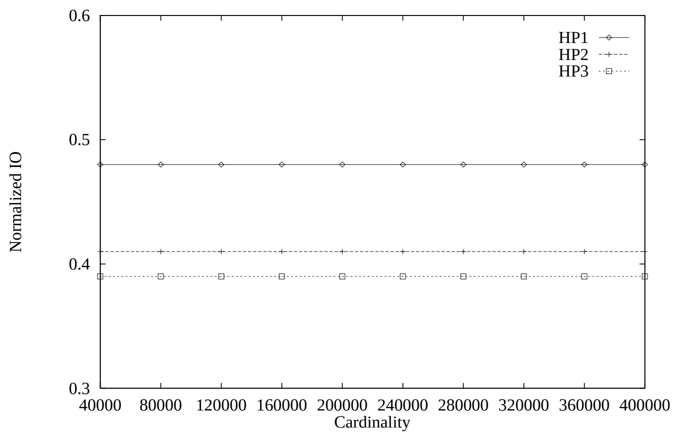

Figure 6: Plot of Normalized IO vs. Cardinality (for number objects per page = 4)

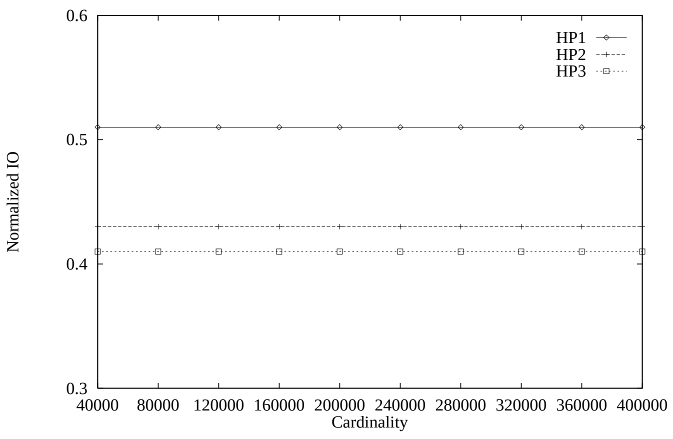

Figure 7: Plot of Normalized IO vs. Cardinality (for number objects per page = 16)

Type 5: Queries Accessing Classes Department & Project	**SEL**
Select Dname From Department Where Proj.Duration $\leq$ 3	0.05
Select DeptId From Department Where Dname = "CS" And Proj.Location = "Hong Kong"	0.5

Type 6: Queries Accessing only Class Project	**SEL**
Select Pname From Project Where Location = "Hong Kong" And Cost $\leq$ 10000	0.05
Select PId, Pname From Project Where Duration $\leq$ 2	0.5
Select Cost() From Project Where Location = "Paris"	0.1

The parameter setting are as follows: the cardinality of the three classes shown in Figure 1 are given by $||Employee||$, $||Department|| = ||Employee|| * FAN_{1,2}$ and $||Project|| = ||Department|| * FAN_{2,3}$ [2]. The length of objects for all the three classes is the same. We note that the number of HCFs of classes *Employee*, *Department*, and *Project* are 2, 2, and 5, respectively.

From the queries, we enumerate all predicates which are: p_1 : Duration $\leq$ 4, p_2 : Cost() [3] $\leq$ 7000, p_3 : Cost() $>$ 7000, p_4 : Location = "Hong Kong", p_5 : Location = "Paris".

4.2 Effect of Varying the Cardinality

In this experiment, we study the impact of variations in the cardinality of the horizontally partitioned class on the performance gain. Parameter values for this experiment shown in Figure 6 and Figure 7 are: cardinality of the class *Project* is varied from 40000 to 400000, page size 4064 bytes, and $FAN_{1,2} = 4$ and $FAN_{2,3} = 1$. All the curves in the plots shown in Figure 6 and Figure 7 are almost horizontal straight lines, showing that the Normalized IO is independent of cardinality of the root class. That is, irrespective of the cardinality of the class we get the same percentage of reduction in number of disk accesses.

[2]We assume that $SHARE_{1,2}$ and $SHARE_{2,3}$ equal 1

[3]Cost() is a method defined on class Project

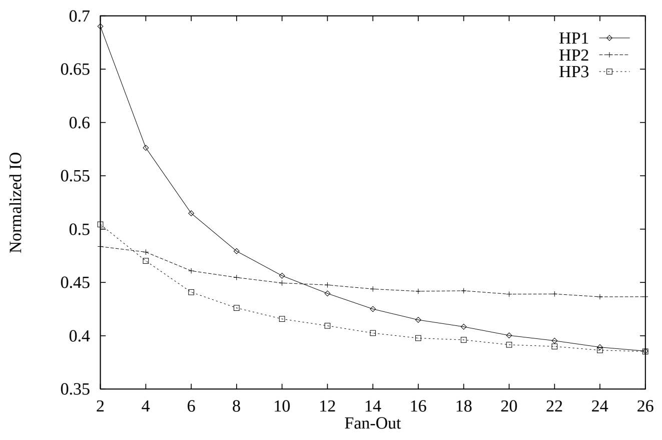

Figure 8: Plot of Normalized IO vs. Fanout (for number objects per page = 4)

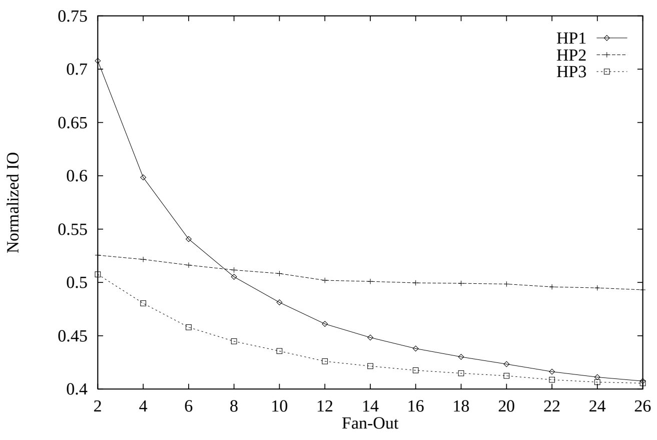

Figure 9: Plot of Normalized IO vs. Fanout (for number objects per page = 16)

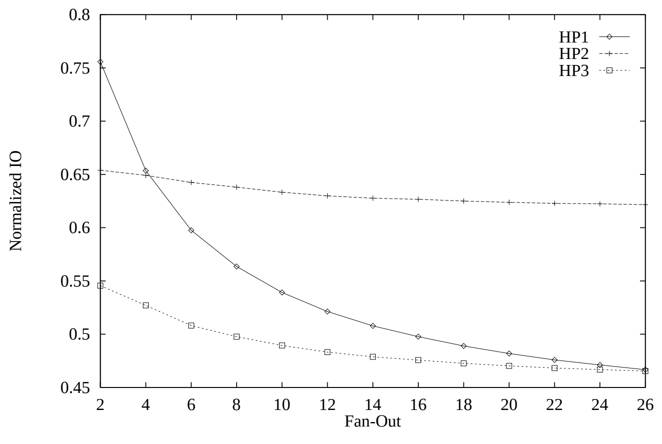

Figure 10: Plot of Normalized IO vs. FAN-OUT (for number objects per page = 64)

4.3 Effect of Varying The Fan-out

In this experiment we study the impact of the variations in the fan-out between the class *Department* and class *Project* on the performance gain. We have varied the values of fan-out in order to see the effect on normalized IO. The parameters values used to produce experimental results shown in Figure 8, Figure 9 and Figure 10 are: cardinality of class *Project* is 1000, page size 4096 bytes. We observe that the performance gain increases with the increase in fan-out. Thus the fan-out has a considerable impact on horizontal class partitioning. We note that the IO normalized is very low when the three classes are horizontally partitioned which means that we can obtain a good performance if all classes in class composition hierarchy are horizontally partitioned.

5 Conclusion and Future Work

In this paper, we have developed an analytical cost model for executing a query in both unpartitioned and horizontally partitioned classes. We used this cost model to show the utility of HCP in reducing the cost of executing queries. Our results show that the HCP can reduce the I/O cost of query execution. This reduction is obtained when all the classes in the class composition hierarchy are horizontally partitioned.

But the HCP may deteriorate the performance of the system due to the overhead which is due to the cost of applying the union operation in order to reconstruct the result of a query. Based on this information, we are working to find out a cost model which takes into consideration this overhead and so as to evaluate its impact on HCP. We are also working on HCFs allocation problem in a distributed object system.

References

[1] J. Banerjee, K Kim, and K. C. Kim. Queries in object oriented databases. *in Proceedings of the IEEE Data Engineering Conference*, February 1988.

[2] L. Bellatreche, K. Karlapalem, and A. Simonet. Horizontal class partitioning in object-oriented databases. *in 8th International Conference on Database and Expert Systems Applications (DEXA'97), Toulouse, Lecture Notes in Computer Science 1308*, pages 58–67, September 1997.

[3] E. Bertino. On modeling cost functions for object-oriented databases. *IEEE Transactions on Knowledge and Data Engineering*, 9(3):500–508, May/June 1997.

[4] E. Bertino and P. Foscoli. An analytical model of object-oriented query costs. *in the Proceedings of the Fifth International Workshop on Persistent Object Systems, San Miniato (Pisa)*, pages 241–261, September 1992.

[5] E. Bertino and C. Guglielmina. Path-index: An approach to the efficient execution of object-oriented queries. *Data & Knowledge Engineering*, 10:1–27, 1993.

[6] 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.

[7] 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.

[8] W. S. Cho, C. M. Park, K. Y Whang, and S. H. So. A new method for estimating the number of objects satisfying an object-oriented query involving partial participation of classes. *Information Systems*, 21(3):253–267, 1996.

[9] 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.

[10] C. W. Fung, K. Karlapalem, and Q. Li. Cost-driven evaluation of vertical class partitioning in object oriented databases. *in Fifth International Conference On Database Systems For Advanced Applications (DASFAA'97), Melbourne, Australia*, pages 11–20, April 1997.

[11] G. Gardarin, J.-R. Gruser, and Z.-H. Tang. A cost model for clustered object-oriented databases. *VLDB*, pages 323–334, 1995.

[12] G. Gardarin, J.-R. Gruser, and Z.-H. Tang. A cost-based selection of path expression processing algorithms in object-oriented databases. *22th International Conference on Very Large Data Bases, VLDB'96*, pages 390–401, 1996.

[13] 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.

[14] 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.

[15] 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.

[16] K. Karlapalem and N. M Pun. Query driven data allocation algorithms for distributed database systems. *in 8th International Conference on Database and Expert Systems Applications (DEXA'97), Toulouse, Lecture Notes in Computer Science 1308*, pages 347–356, September 1997.

[17] W. Kim. A model of queries for object-oriented databases. *in Proceedings of the 15th. International Conference on Very Large Databases (VLDB'89)*, August 1989.

[18] D. Maier, G. Graefe, L. Shapiro, S. Daniels, T. Keller, and B. Vance. Issues in distributed object assembly. In *Distributed Object Management*, pages 179–181. Morgan Kaufman Publishers Inc., 1994.

[19] 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.

[20] S.B. Navathe and M. Ra. Vertical partitioning for database design : a graphical algorithm. *ACM SIGMOD*, pages 440–450, 1989.

[21] C. Ozkan, A. Dogac, and M. Altinel. A cost model for path expressions in object-oriented queries. *Journal of Database Management*, 7(3):25–33, 1996.

[22] M. T. Özsu and P. Valduriez. *Principles of Distributed Database Systems.* Prentice Hall, 1991.

[23] D. D. Straube and M. T Özsu. Queries and query processing in object oriented database systems. *ACM Transactions on Information Systems*, 18(4):387–430, October 1990.

[24] S. B. Yao. Approximating the number of accesses in database organizations. *Communication of the ACM*, 20(4):260, April 1977.