The VLDB Journal (2003) 12: 187–210 / Digital Object Identifier (DOI) 10.1007/s00778-002-0084-7

Cost-driven vertical class partitioning for methods in object oriented databases

Chi-Wai Fung[1], **Kamalakar Karlapalem**[2], **Qing Li**[3]

[1] Department of Information & Communications Technology, HK Institute of Vocational Education (Tuen Mun), Tuen Mun, Hong Kong, China; e-mail: cwfung@vtc.edu.hk
[2] International Institute of Information Technology, Gachhilbowli, Hyderabad 5000019, India; e-mail: kamal@iiit.net
[3] Department of Computer Engineering and Information Technology, City University of Hong Kong, Tat Chee Avenue, Kowloon, Hong Kong, China; e-mail: itqli@cityu.edu.hk

Edited by M.H. Scholl. Received: March 29, 1999 / Accepted: March 11, 2002
Published online: April 3, 2003 –

Abstract. In object-oriented databases (OODBs), a method encapsulated in a class typically accesses a few, but not all the instance variables defined in the class. It may thus be preferable to vertically partition the class for reducing irrelevant data (instance variables) accessed by the methods. Our prior work has shown that vertical class partitioning can result in a substantial decrease in the total number of disk accesses incurred for executing a set of applications, but coming up with an optimal vertical class partitioning scheme is a hard problem. In this paper, we present two algorithms for deriving optimal and near-optimal vertical class partitioning schemes. The cost-driven algorithm provides the optimal vertical class partitioning schemes by enumerating, exhaustively, all the schemes and calculating the number of disk accesses required to execute a given set of applications. For this, a cost model for executing a set of methods in an OODB system is developed. Since exhaustive enumeration is costly and only works for classes with a small number of instance variables, a hill-climbing heuristic algorithm (HCHA) is developed, which takes the solution provided by the affinity-based algorithm and improves it, thereby further reducing the total number of disk accesses incurred. We show that the HCHA algorithm provides a reasonable near-optimal vertical class partitioning scheme for executing a given set of applications.

Keywords: Vertical class partitioning – Object-oriented databases – Method-induced – Cost-driven – Affinity-based – Analytical cost model – Hill-climbing heuristic algorithm

1 Introduction

Database design plays an important role in supporting end-user applications. Some database design techniques, such as conceptual schema design, view integration, and schema normalization, aim at capturing appropriate integrity constraints and complete incorporation of database application requirements, whereas techniques such as index selection and database clustering aim towards efficiently executing end-user applications. Vertical class partitioning is a technique for facilitating efficient execution of end-user applications by reducing irrelevant instance variable (attribute) access. Although this problem has been addressed in relational database systems, there has been very little work done on vertical partitioning in object-oriented database (OODB) systems. Part of the reason for this is due to the complexity of OODB models supporting subclass hierarchy and class composition hierarchy, which complicate the definition and representation of vertical partitioning. Another reason for the complexity is due to the encapsulation property of the OODB model, which enforces accesses to objects through encapsulated methods.

1.1 Motivation for vertical class partitioning

(1) In OODB systems, methods encapsulated in a class typically access a few, but not all, of the instance variables defined in the class. Therefore, a judicious vertical class partitioning of the class can drastically reduce the irrelevant instance variables accessed by the methods and improve the performance. In the case of query processing, most OODB query languages allow for method invocation in the *project* and *select* clauses, and only some, but not all, instance variables are accessed.
(2) Though the encapsulation feature of OODB systems complicates the partitioning problem, it has been demonstrated as being useful for providing fragmentation transparency support [18,19], thus enabling easier development of applications and their maintenance.
(3) The problem of creating an optimal vertical class partitioning scheme is a hard problem [26], hence heuristic algorithms need to be designed to come up with near-optimal vertical class partitioning schemes. Two main heuristics, namely, affinity-based [22] and cost-driven [5], have been proposed in prior research.
(4) With the presence of methods in OODB systems, the cost-driven approach requires the development of a cost model

This research has been supported, in part, by Hong Kong UGC Research Grants Council under grant CityU 733/96E and CityU 1119/99E.

for method execution. This is a hard and relevant problem by itself, and is addressed in this paper.

(5) For the next generation database applications, such as document management, multimedia and hypermedia applications, many of the instance variables tend to be of very large-sized objects, and should not be retrieved if they are not accessed by the applications. Vertical class partitioning techniques can be applied to these applications to provide faster access to the relevant data based on user access characteristics.

1.2 Related work in vertical class partitioning

There has been some preliminary work done in recent years on class partitioning for OODBs. In particular, the basic ideas about class partitioning in OODBs were developed in [6, 17, 18, 19, 21]. In [21], the issues involved in distributing OODBs were identified, with a main emphasis on the factors that affect the distribution and partitioning for OODBs. In [6], the authors presented an approach to vertical partitioning of OODBs, which is based on the concept of affinity [22]. However, there was no representation scheme provided for vertical fragments. In addition, the physical costs corresponding to the savings in the amount of irrelevant data accessed and the overhead incurred due to vertical partitioning were ignored.

In [17,18,19], representation schemes for vertical class fragments have been presented, along with a *method-induced methodology for class partitioning* and *the support for fragmentation transparency*. In [8,9,10], we developed a cost model based on [11] for query execution in vertically partitioned OODBs. Based on the cost model, we reported on the analytical evaluation of vertical partitioning under different environments, namely: (1) different numbers of fragments; (2) different fan-outs in class composition hierarchy; and (3) different cardinalities for classes. We showed that the cost-based approach is superior to the affinity-based approach in that it guarantees the minimum number of disk accesses to process all the queries. We note that the main themes of [8,9] were focussed on the analytical evaluation of the cost model and the utility of vertical class partitioning under different OODB system environments; in addition, the cost model was based on query execution. Since the data objects in an OODB are accessed by encapsulated methods, there is also a need to obtain the cost of executing methods for evaluating the effectiveness of vertical class partitioning.

1.2.1 Partitioning vs clustering, indexing and complex object assembly

Vertical partitioning is a facility in OODBs to reduce the number of disk IOs for application execution by reducing the accesses to irrelevant instance variables. In contrast, indexing in OODBs is a facility to reduce the number of disk IOs in query execution (see, for example, [2,20]) by reducing the accesses to irrelevant object instances (as compared with sequential scanning). In other words, indexing reduces disk IOs at the *object instance level*, but since typically not all the instance variables are relevant to a specific query being processed, access to irrelevant instance variables can still occur, whereas vertical partitioning reduces disk IOs at the *instance variable level*.

In [14], an "assembly" operator is devised to address the problem of avoiding "pointer chasing on disk". This assembly operator is designed to improve the processing of bulk data types such as sets in object bases. When compared with vertical partitioning, complex object assembly concentrates on the *physical object instance level*, whereas vertical partitioning concentrates on the *instance variable level*. During query processing, complex object assembly still accesses irrelevant instance variables. If the irrelevant instance variables are large, complex object assembly may not be very useful.

We note that partitioning is a logical database design technique whereas clustering/indexing/complex object assembly is a physical database design technique. The use of indexing is complementary and orthogonal to the use of partitioning. That is, once the classes are vertically partitioned, indices can be built to execute the applications more efficiently. Since vertical class fragments redefine the OODB schema with new classes, clustering and complex object assembly can be applied to further improve efficient execution of applications. The issue of integrating clustering, indexing, and complex object assembly with vertical partitioning is, however, beyond the scope of this paper.

1.3 Related work in the method execution cost model

In an OODB, accesses to data are through encapsulated methods. The encapsulation of methods forces the user to access the data by invoking the methods and keeps the user from making use of knowledge about the data structures used to store the objects or the implementation of the method. The encapsulation property of methods in the OODB makes the estimation of execution cost more difficult than conventional query processing in relational databases. Furthermore, the encapsulation of methods also complicates the issue of query optimization. In some early studies of OODB query optimization, methods are not considered in query optimization. However, some systems overcome this difficulty by treating the query optimizer as a special application that can break encapsulation and access information directly [3]. However, in order to come up with optimal method execution plans, an analytical cost model for method execution is essential.

In other previous research, encapsulated methods in OODBs are treated as black boxes and are neglected from cost analysis. There has been very little discussion on the cost model for general method execution, due to the complexity of the detailed semantics of a general method. Unlike relational databases, query processing in OODBs has to deal with methods whose different types of invocation can affect the cost model. We consider three types of invocations found in general purpose programming languages: simple invocation, conditional invocation, and repeated invocation. To utilize this model, run-time invocation frequencies are collected through monitoring the past runs. The novelty of this approach is the use of methods of OODBs as input for describing the application load on each instance variable. Method semantics can be abstracted by means of graphs describing the flow of execution with control structures.

The following is a summary of some related work on OODB method execution cost models:

(1) In [28], the optimization of queries containing methods is discussed. Some new strategies of object query optimization (for the classification and evaluation of selections and joins containing path expressions or methods) are proposed. Object query graphs are used to represent object queries and to capture different kinds of selections and joins. Object query graphs are also used to factorize common sub-path expressions among not only path expressions in queries, but also maximal common sub-path expressions in methods and to generate query evaluation plans.
(2) In [27], no specific cost model is proposed. Instead, they assume that the OODBMS is capable of using OID-stream statistics to derive a cost for method calls. Appropriate OID-stream statistics might be stream cardinality and information about the classes represented in the stream. For a given method call, the OODBMS could derive a processing cost and statistics for the resulting output OID-stream.
(3) In [13], a model is developed for measuring the cost of a predicate function (similar to our notion of method). The cost of a predicate function is computed by adding up the costs for each sub-predicate function in the predicate function expression. Given a predicate function $p(a_1, \ldots, a_n)$, the expense per object is recursively defined as:

$$e_p = \begin{cases} \sum_{i=1}^{n} e_{a_i} + percall\_cpu(p) + \\ \quad perbyte\_cpu(p) * (byte\_pct(p)/100) * \\ \quad\quad \sum_{i=1}^{n} bytes(a_i) + access\_cost \\ \quad \text{if } p \text{ is a predicate function} \\ 0 \quad \text{if } p \text{ is constant or instance variable} \end{cases}$$

and e_{a_i} is the recursively computed expense of argument a_i. The parameter values as listed in Table 1 are initial estimations based on either default values or system statistics. After repeated applications of a predicate function, one could collect performance statistics and use curve-fitting technique to make estimates about the predicate function's behavior.
(4) The HERMES project is an ESPRIT joint project [12]. The objective of the project is to develop principles and methodologies for the design and implementation of high-performance integrated system environments for the retrieval of real-time, delay-sensitive, and synchronized multimedia data. One of their contributions is their experimentation on method execution cost calibration and history.

1.4 Contributions and paper organization

In contrast, the main contributions of our work described in this paper are as follows:

(1) A cost model for method execution in OODBs is developed. This cost model is used in the subsequent algorithms to generate the optimal vertical class partitioning scheme. To the best of our knowledge, this is the first piece of work on developing a cost model for complex method execution in OODBs.
(2) A cost-driven algorithm (CDA) is developed, which guarantees to produce the cost-optimal partitioning scheme based on exhaustive enumeration and has a high computational complexity O(n^n) [4], where n is the number of instance variables in the class.
(3) An affinity-based algorithm which has low computational complexity of O(n^2) [24] is designed. Furthermore, it is shown that the affinity-based algorithm does not necessarily generate the optimal vertical class partitioning scheme.
(4) Finally, a hill-climbing heuristic algorithm (HCHA) based on both the cost-based and affinity-based approaches is designed. This algorithm uses the initial solution generated by affinity-based algorithm and incrementally evolves it to generate optimal or near optimal vertical class partitioning scheme.

The rest of the paper is organized as follows: Sect. 2 presents the characteristics and specifications of OODB methods. Section 3 presents an analytical cost model for method-induced vertical partitioning. Section 4 presents two different vertical partitioning algorithms, namely, CDA and HCHA, and Sect. 5 presents analytical evaluation results on these algorithms. Section 6 presents the empirical validation of the cost model and discussions on the applicability and limitations. Conclusions and future work are given in Sect. 7.

2 Method characteristics and specification

To lay down a foundation for the development of a cost model for method execution and method-induced vertical class partitioning (MI-VCP), in this section we first review a core object-oriented data model. Through the concept of method dependency graph, a general model for method execution with emphasis on the different types of method invocations is developed.

2.1 A core object-oriented data model

To make our discussions general-purpose, we focus on the basic concepts that are mandatory and/or common to most OODB models and systems [1,15,16]. These elementary concepts also form the core of the OODB model that we shall assume for our subsequent discussion.

Table 1. Method execution cost terms

Term	Descriptions
percall_cpu	execution time per invocation, regardless of the size of the arguments
perbyte_cpu	execution time per byte of arguments
byte_pct	percentage of arguments bytes that the function will need to access
bytes	expected (return) size of the argument in bytes
access_cost	cost of retrieving any data necessary to complete the function

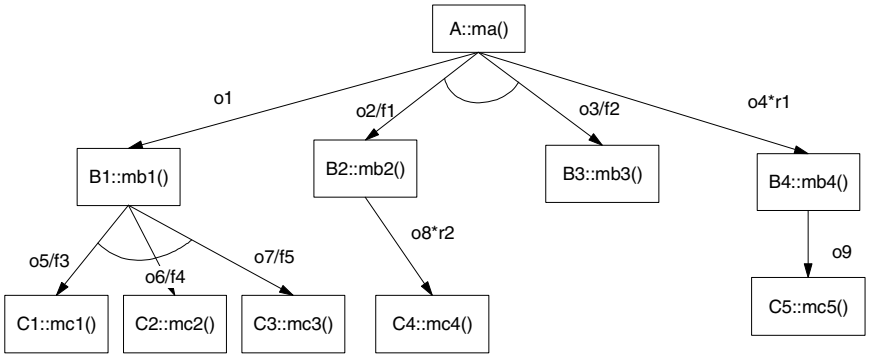


Fig. 1. MDG of an example method *ma()*

The most fundamental concepts of an OODB include class, inheritance, and object identifier (OID). A class defines a set of *instance variables* (or attributes) that constitute the "state", and a set of procedural *methods* that embody the "behavior" of its objects. Classes are organized into an inheritance (Is-A) hierarchy, in which a subclass *inherits* the attributes and methods defined in the superclass(es) for its objects. For each attribute, the set of values it may have is confined by its class type (which is called the domain class); both atomic (e.g., integer, string of characters) and abstract (i.e., object) domain classes are possible for an object. Objects are uniquely distinguished with their (system-generated) object identifiers (OIDs), hence the existence of an object is independent of its state (i.e., attribute values). Besides the Is-A hierarchy, OODBs exhibit another form of useful hierarchy called the composition hierarchy, which captures the "Is-Part-Of" relationship between a parent class and its component classes.

A method has a signature (interface) including the method's name, a list of parameters, and a list of return values. Parameters and return values may be either value-based or object-based instance variables (VBIVs or OBIVs). Methods are inherited from the superclass(es). A subclass may alter the method code of an inherited method or introduce additional methods.

A simple method does not call/invoke any other method. A complex method [6] calls/invokes other methods. A method that accesses VBIVs only accesses the leaf node of a composition hierarchy. On the other hand, a method that accesses OBIVs can potentially invoke other methods. A method can return atomic values (such as real, integers and string) or return object identifiers.

2.2 Method dependency graph

Method dependency graph (MDG) is a graphical representation of a complex method which calls/invokes other methods [18,19]. Each method dependency graph represents the behavior of a complex method accessing an *object-based instance variable* (OBIV). A MDG has a set of nodes and a set of directed edges. The nodes represent the methods that are invoked by the complex method, and the edges denote the sequence in which the methods are invoked. Since a complex method can invoke other methods (both simple and complex), a complex method can be naturally represented by a set of MDGs. An example MDG is illustrated in Fig. 1.

2.2.1 Method execution semantics

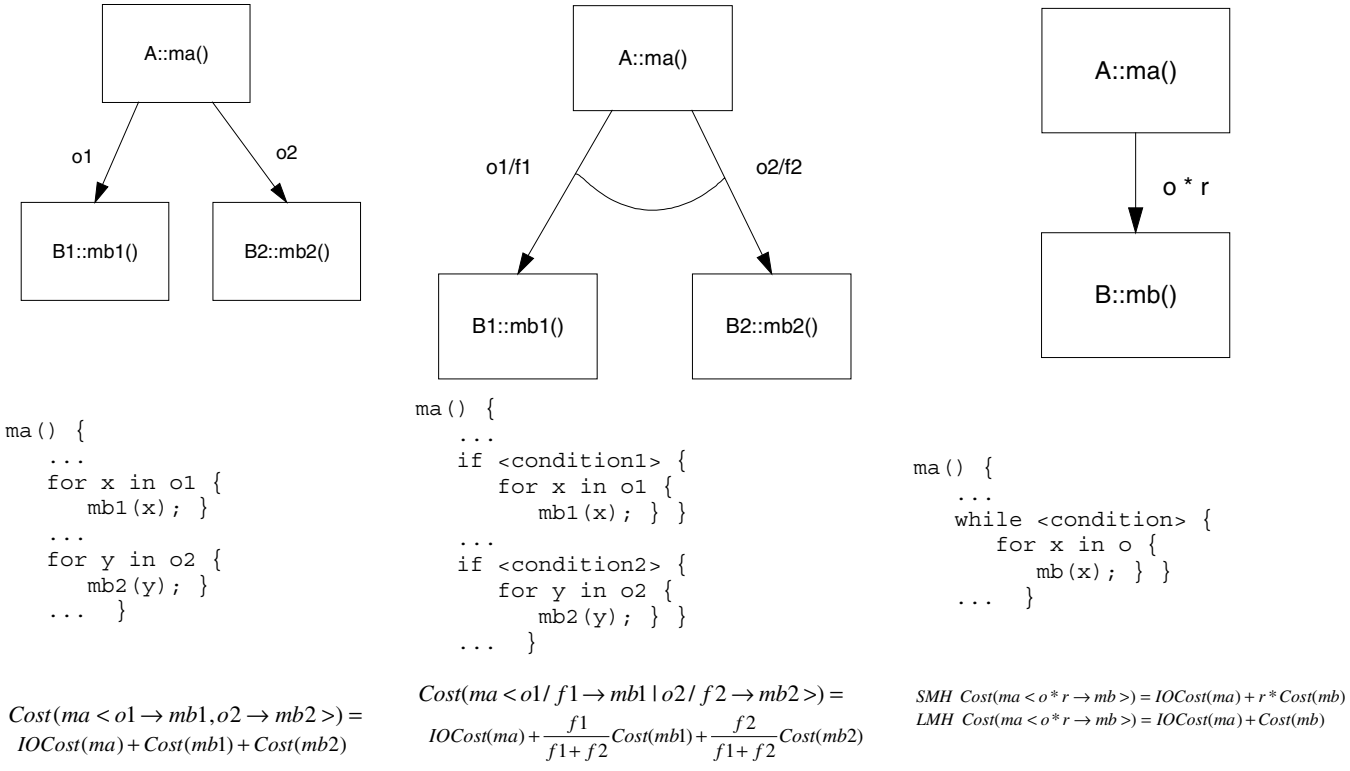
In most of the previous research, methods are treated as "black boxes" and are neglected from cost analysis. Nevertheless, an analytical cost model for method execution in OODBs is a problem relevant to class partitioning. We tackle this problem by studying the cost relationship between different subprograms defined in a method. As many query languages do not (yet) support recursion, we focus on non-recursive methods in this paper.

The following different types of method invocation are distinguished: (1) simple invocation; (2) conditional invocation; and (3) repeated invocation. We discuss the different types of method invocation below, the purpose of which is to yield the basic cost formula for each type. For expository purpose, the discussion is based on the three example methods illustrated in Fig. 2.

Method invocation dependency (MID) terms

In Fig. 2a, *ma()* defined in class A is a complex method invoking methods *mb*1()and *mb*2()in sequence. The two OBIVs $o1$ and $o2$ are defined in class A, with the domain $A.o1$ being of class $B1$ and $A.o2$ being of class $B2$. We introduce a term of *method invocation dependency* (MID) as $ma < o1 \rightarrow mb1\ ,\ o2 \rightarrow mb2 >$ which, intuitively captures the MDG information of *ma()* in that the methods *mb1()* and *mb2()* are called *directly* by method *ma()* (i.e., they are inside the "<>" which contains the methods called by *ma()*). The notation $< o1 \rightarrow mb1\ ,\ o2 \rightarrow mb2 >$ implies that methods *mb1()* and *mb2()* are executed one after the other in sequence, and the notation $o1 \rightarrow mb1$ implies that the sub-method *mb1()* in the MDG is invoked by *ma()* through an object-based instance variable $o1$. The IO cost of the method *ma()* is calculated by a summation of the IO cost of executing the two sub-methods *mb*1()and *mb*2(),plus the IO cost of executing the method *ma() without regard* to the cost of executing *mb*1()and *mb*2() *(as they are already considered)*. Figure 2a also shows the resultant cost formula in this case. The details of *IOCost* and *Cost* will be described in Sect. 2.3 and Sect. 3.2, respectively.

In Fig. 2b, the complex method *ma()* is assumed to invoke methods *mb*1()and/or *mb*2() conditionally. That is, the method *ma()* branches out to method *mb1()* or *mb2()*. The arc connecting branches in the MDG represents a conditional execution. Furthermore, the method *mb*1() is to be invoked with relative frequency $f1$ and the method *mb*2() to be invoked with



a **b** **c**

Fig. 2a–c. MDG, sample code, and cost formula for **a** simple method invocation, **b** conditional method invocation, and **c** repeated method invocation

relative frequency $f2$. Similar to [27], we assume that the underlying OODBMS can monitor the method execution to keep track of these execution frequency values (whenever required by the database administrator). This leads to an extra overhead to the OODBMS but will be amortized by the higher-quality cost estimation produced. Note that per method invocation of *ma()*, both of its sub-methods may be executed inclusively or only one of them is actually executed. We therefore introduce a new MID term: $ma < o1/f1 \rightarrow mb1 \mid o2/f2 \rightarrow mb2 >$, which indicates that method *ma()* will execute either method *mb1()* (with frequency $f1$) and/or method *mb2()* (with frequency $f2$); the "|" inside the "[ ]" means a conditional execution between the two (or more) methods. The IO cost of method *ma()* is calculated by the cost formula as shown in the lower portion of Fig. 2b.

To facilitate the subsequent discussion, as in [11], we make the following observations regarding the amount of main memory, *Large Memory Hypothesis (LMH)* – the main memory size is so large that we have enough memory buffers for all the incoming objects (i.e., in loading objects from the disk, they are only loaded once). *Small Memory Hypothesis (SMH)* – the main memory size is so small that we can afford to allocate only one page of memory buffer for each class or fragmented class (i.e., during method execution, the same objects or object fragments of a particular class are required to be loaded into the main memory multiple times and cause a high increase in the number of disk accesses). Note that LMH is the best case and the SMH is the worst case scenario with respect to the number of disk accesses required to execute a method.

As shown in Fig. 2c, the complex method *ma()* in this case repeatedly invokes method *mb()* with a repeating factor r (i.e., *mb()* is executed for r times). We introduce one more MID term: $ma < o * r \rightarrow mb >$. This term depicts that method *ma()* will execute method *mb()* repeatedly (with a repeating factor r); the "*" inside the "< >" means a repeated execution. For SMH, the IO cost of method *ma()* is as shown by the upper cost formula in Fig. 2c, where the cost of method *mb()* is multiplied by the repeating factor r. Again, similar to [27], we assume that the underlying OODBMS can monitor the method execution to keep track of these repeating factor values. This formula is based on the assumption that the objects required by method *mb()* are repeatedly loaded into the main memory each time they are needed (i.e., the SMH case). If we have enough main memory to hold all the objects involved (i.e., the LMH case), then the objects required by method *mb()* need not be repeatedly loaded into the main memory. Hence the IO cost of method *ma()* in this case is as shown by the lower cost formula in Fig. 2c, where we see the multiplying factor r disappears as a consequence.

2.3 Method execution cost model

Let $m < q_1, q_2, \ldots, q_n >$ be the MDG representation of a method $m()$,where q_1, q_2, …, q_n are the *MID* terms (as introduced in Sect. 2.2.1), then the cost of executing $m()$ is similar to that of [13], that is:
If q_i is of form $o_i \rightarrow m_i$ then

Cost(q_i)=Cost(m_i).

If q_i is of form $[o_j/f_j \rightarrow m_j \mid \ldots \mid o_k/f_k \rightarrow m_k]$ then

$$Cost(q_i) = \frac{f_j}{f_j + \ldots + f_k} Cost(m_j) + \ldots + \frac{f_k}{f_j + \ldots + f_k} Cost(m_k).$$

If q_i is of form $o_i * r_i \rightarrow m_i$ then

Cost(q_i) $= r_i$*Cost(m_i)* in the case of Small Memory Hypothesis (SMH) or

Cost(q_i) = *Cost(m_i)* in the case of Large Memory Hypothesis (LMH).

Similarly *Cost(m_i)*, *Cost(m_j)* and *Cost(m_k)* are recursively defined. In conclusion, a nested method invocation can be represented as a combination of the above types of method invocation. We now illustrate method execution cost calculation by means of an example.

Example. Consider the method *ma()* as shown in Fig. 1, it can be represented by using MID terms as follows:

$$ma < o1 \rightarrow mb1,\ [o2/f1 \rightarrow mb2 \mid o3/f2 \rightarrow mb3],\ o4 * r1 \rightarrow mb4 >$$

Assuming in SMH cases, the cost of executing *ma()* is calculated as follows:

$$\begin{aligned} &Cost(ma < o1 \rightarrow mb1,\ [o2/f1 \rightarrow mb2 \mid o3/f2 \rightarrow mb3],\ o4 * r1 \rightarrow mb4 >) \\ &= IOCost(ma) + Cost(mb1 < [o5/f3 \rightarrow mc1 \mid o6/f4 \rightarrow mc2 \mid o7/f5 \rightarrow mc3] >) \\ &+ \frac{f1}{f1+f2} Cost(mb2 < o8 * r2 \rightarrow mc4 >) \\ &+ \frac{f2}{f1+f2} Cost(mb3) \\ &+ r1 * Cost(mb4 < o9 \rightarrow mc5 >). \end{aligned}$$

In the above formula,

$$\begin{aligned} &Cost(mb1 < [o5/f3 \rightarrow mc1 \mid o6/f4 \rightarrow mc2 \mid o7/f5 \rightarrow mc3] >) = IOCost(mb1) \\ &+ \frac{f3}{f3+f4+f5} Cost(mc1) + \frac{f4}{f3+f4+f5} Cost(mc2) \\ &+ \frac{f5}{f3+f4+f5} Cost(mc3), \end{aligned}$$

$$Cost(mb2 < o8 * r2 \rightarrow mc4 >) = IOCost(mb2) + r2 * Cost(mc4),$$

and

$$Cost(mb4 < o9 \rightarrow mc5 >) = IOCost(mb4) + Cost(mc5),$$

with

$$Cost(mb3) = IOCost(mb), \quad Cost(mc1) = IOCost(mc1),$$
$$Cost(mc2) = IOCost(mc2), \quad Cost(mc3) = IOCost(mc3),$$
$$Cost(mc4) = IOCost(mc4),$$
$$\text{and } Cost(mc5) = IOCost(mc5).$$

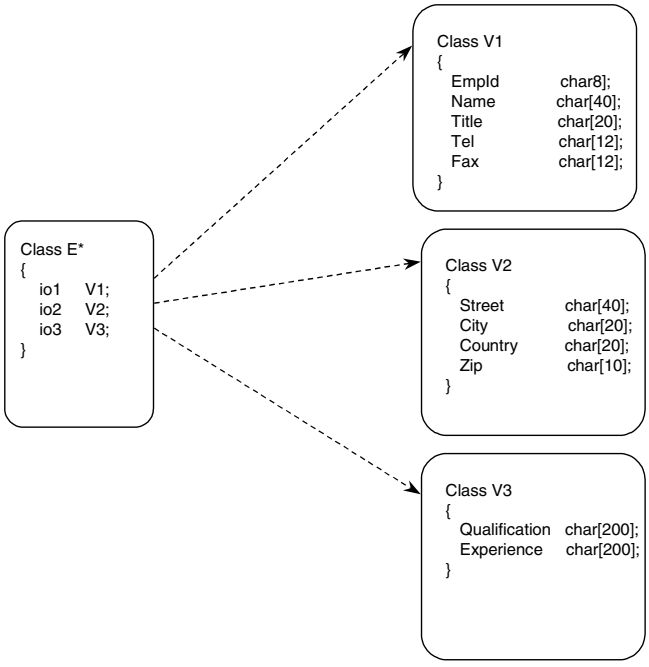


Fig. 3. Vertical class fragments *V1, V2, V3* and composite object E^* of class *Employee*

2.4 Estimation of parameter values

To enable us to calculate the cost of executing a complex method, we need to have estimates on the following method invocation parameters: (1) the relative frequencies between different sub-methods in conditional invocation; and (2) the repeating factor for repeated invocation. These estimates can be obtained from any one or all of the following sources:

(a) extra counter variables may be attached to each method to obtain the method invocation parameter values;
(b) the underlying OODBMS monitors and keeps track of statistics on these method invocation parameter values.
(c) the method designer provides a preliminary estimate.

With some extra overhead, accurate parameter values can also be obtained from (a) and (b). The extra overhead of monitoring/keeping track of the query/method execution in (a) and (b) is amortized by the performance gain after we apply the method-induced vertical class partitioning (VCP) technique. In (c), the parameter values are estimated by the VCP designer, hence the values will not be too accurate and can affect the utility of VCP. However, our HCHA provides guidelines for choosing the initial VCP scheme. The VCP scheme can be further refined when more accurate parameter values are subsequently obtained.

3 Cost model

In this section, we present an analytical cost model for method-induced vertical partitioning in OODBs. First, we consider an example class (*Employee*) which has been partitioned based on a vertical partitioning scheme {*V1, V2, V3*}. As shown in Fig. 3, the original class *Employee* can be internally represented by a class E^* with a set of object-based instance variables: *io1,*

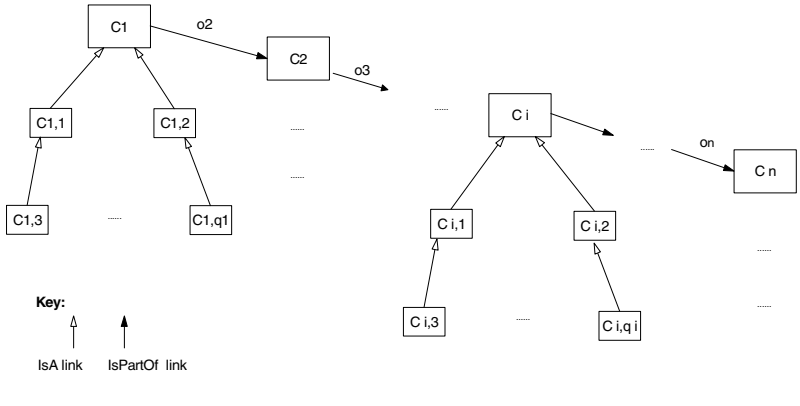


Fig. 4. Example schema for a MDG path in unpartitioned case

io2 and *io3*, where each object-based instance variable refers to an object from vertical fragment *Vj* (for $1 \leq j \leq 3$). This embodies an object-oriented representation of a vertical partitioning of a class [17,18]: each of the vertical class fragments is represented as a class, and a logical object of class *Employee* is internally represented as a composite object (of class E^*) that consists of objects from the vertical class fragments. As vertical class fragments are also represented as classes, this uniform/homogeneous approach has the added advantage of supporting fragmentation transparency [18,19]. Furthermore, this approach does not require extra support in schema maintenance (as found in vertical partitioning for relational databases). Thus, in the remaining discussions this internal representation of vertical partitioned classes is assumed.

3.1 Basic concepts

Similar to other related work (e.g., [11]) on cost models in OODBs, we have the cost model parameters as shown in Table 2. Some of the parameters can readily be obtained from the system catalog (like sizes of instance variables), while others require the OODBMS to keep track of their values, such as the fan-outs between classes in the class composition hierarchy.

The total cost to execute a method is calculated as:

Total_cost = *Disk_IO_cost* + *CPU_cost*,

where *Disk_IO_cost* is the cost of performing disk IO and *CPU_cost* is the cost for performing the computation during the method execution. As in [5], we concentrate in this paper on the *Disk_IO_cost* and disregard the *CPU_cost*. This is because for very large database applications with huge amount of data accesses, the *CPU_cost*'s contribution towards the *Total_cost* would be insignificant.

To facilitate the building of the cost model, we define the following types of path expressions (with the last two being incorporated from [28]):

- *MDG path expression* – a path expression obtained from the MDG. An example is $A \cdot o1 \cdot o5$ of Fig. 1. Note that a MDG can contain more than one MDG path expression.
- *Parameter path expression* – a path expression originating from parameter objects of a method.
- *Hidden path expression* – all path expressions that are not MDG or parameter path expressions.

A particular MDG path (i.e., one of the several possible MDG paths in a MDG) with path length n is of the general form $C_1 \cdot o_2 \cdot o_3 \cdot \ldots \cdot o_n$, where C_1 is the root class of the MDG, and $o_i \in C_{i-1}$ (i.e., o_i is the OBIV defined in class C_{i-1} with a domain of class C_i). An example schema of a MDG path with a class inheritance hierarchy is shown in Fig. 4. For convenience, we denote the k the subclass of a class inheritance hierarchy rooted at class C_i by the notation $C_{i,\,k}$, where k ranges over 1 through q_i (which is the total number of subclasses of class C_i). To make the cost formulae more compact, we can also denote the root class C_i as $C_{i,\,0}$. We further denote the jth vertical fragment of the kth subclass of class C_i by $C_{i,\,k}V_j$, where j ranges over 1 through m_i (which is the total number of fragments for every class in the class inheritance hierarchy rooted at class C_i). For simplicity, we assume after vertical partitioning, all the class/subclasses of the whole class inheritance hierarchy are partitioned into the same number of fragments. An example schema for a MDG path with vertical partitioning is shown in Fig. 5. Similar notation is also used for hidden and parameter paths.

The total number of pages occupied by a class collection (i.e., class extent) C is given by: $|C| = \left\lceil \frac{\|C\| * SC}{PS} \right\rceil$, where $\lceil\ \rceil$ is the ceiling function and *SC* is the object size. When applying these formulae to a subclass hierarchy, we assume that within the same class/subclass, objects are stored together. Between different subclasses, however, objects are stored independently. This means that all the subclasses are not clustered into one huge class collection for the reason of efficient processing of queries on individual subclasses. When we apply these formulae to a subclass hierarchy with vertical fragments, we further assume that the fragments of different classes/subclasses are stored independently (for the same reason as the unpartitioned case).

In evaluating a predicate, it is important to estimate the number of page accesses to a class collection during path expression traversal. In estimating the number of page accesses to a collection, we use the Yao function [29]. Specifically, given n records uniformly distributed into b blocks or pages ($1 < b \leq n$), if k records ($k \leq n$) are randomly selected from the n records, the expected number of page accesses is given by $Yao(n,\, b,\, k) = b * \left[1 - \prod_{i=1}^{k} \frac{nd-i+1}{n-i+1}\right]$, where $d = 1 - 1/b$. (The expected number of page accesses is not equal to k because some pages may contain two or more records.) Note that the Yao function models database accesses with limited buffer cache. In our discussion, we usually take $n = \|C\|$, $b = |C|$ and $k = SEL * \|C\|$ (where SEL is the selectivity of the

Table 2. Cost model parameters for MI-VCP

Category	Parameter	Meaning
Database	$\|\|C_{i,k}\|\|$	cardinality of class collection $C_{i,k}$(i.e., kth subclass of ith class along the class composition hierarchy)
	$\|C_{i,k}\|$	number of pages occupied by class $C_{i,k}$
	$SC_{i,k}$	size of object (in unit of byte) in class $C_{i,k}$
	q_i	number of subclasses in the class inheritance hierarchy rooted by class C_i
	$fan_{i-1,\,i,\,j,\,k}$	fan-out for the class composition hierarchy from jth subclass of class C_{i-1} to the kth subclass of class C_i
	n	path length of the MDG path, i.e., the number of classes along the MDG path in the class composition hierarchy
	$NP_{i,\,k}$	number of objects (of kth subclass of class C_i) per page. If $SC_{i,k} < PS$ then $NP_{i,\,k}$ is $\left\lfloor \frac{SC}{SC_{i,k}} \right\rfloor$, otherwise we set it to 1
	h	path length of the hidden path
	p	path length of the parameter path
	b	B$^+$-tree index average fan-out
	PS	page size of the file system (in unit of byte)
Method	$M_{i,\,j,\,k}$	a binary variable, it is of value 1 if the method accesses jth vertical fragment of the kth subclass in class C_i; 0 otherwise
	$Sproj_{\,i,\,k}$	length of output result that is within kth subclass in class C_i
	$Sproj_{\,i,\,j,\,k}$	length of output result that is within jth fragment of the kth subclass in class C_i
	$ref_{i,\,k}$	number of object references for kth subclass in class C_i during the path expression evaluation process along the class composition hierarchy in the MDG path
	SEL_i	selectivity of the method's predicate on class C_i
Vertical Partitioning	m_i	number of fragments in class C_i
	$SC_{i,\,k}\,V_j$	size of jth fragment in kth subclass in class C_i
	$\|C_{i,\,k}\,V_j\|$	number of pages occupied by jth fragment of the kth subclass of class C_i
	VP_i	a binary variable, it is of value 1 if class/subclasses of C_i is vertically partitioned; 0 otherwise
	$\|CO_{i,\,k}\|$	number of pages occupied by the composite objects of class $C_{i,k}$
		<For the hidden path, the following parameters are similarly defined: $\|\|H_{i,\,k}\|\|$, $\|H_{i,\,k}\|$, $SH_{i,\,k}$, hq_i , $Hfan_{\,i-1,\,i,\,j,\,k}$, $HM_{\,i,\,j,\,k}$, $Href_{i,\,k}$, $HSel_i$, hm_i, $SH_{i,\,k}V_j$, $\|H_{i,\,k}V_j\|$, HVP_i, $\|HCO_{i,\,k}\|$. For the parameter path, the following parameters are similarly defined:$\|\|P_{i,\,k}\|\|$, $\|P_{i,\,k}\|$, $SP_{i,\,k}$, pq_i, $Pfan_{\,i-1,\,i,\,j,\,k}$, $PM_{\,i,\,j,\,k}$, $Pref_{i,\,k}$, $PSel_i$, pm_i, $SP_{i,\,k}V_j$, $\|P_{i,\,k}V_j\|$, PVP_i, $\|PCO_{i,\,k}\|$. >

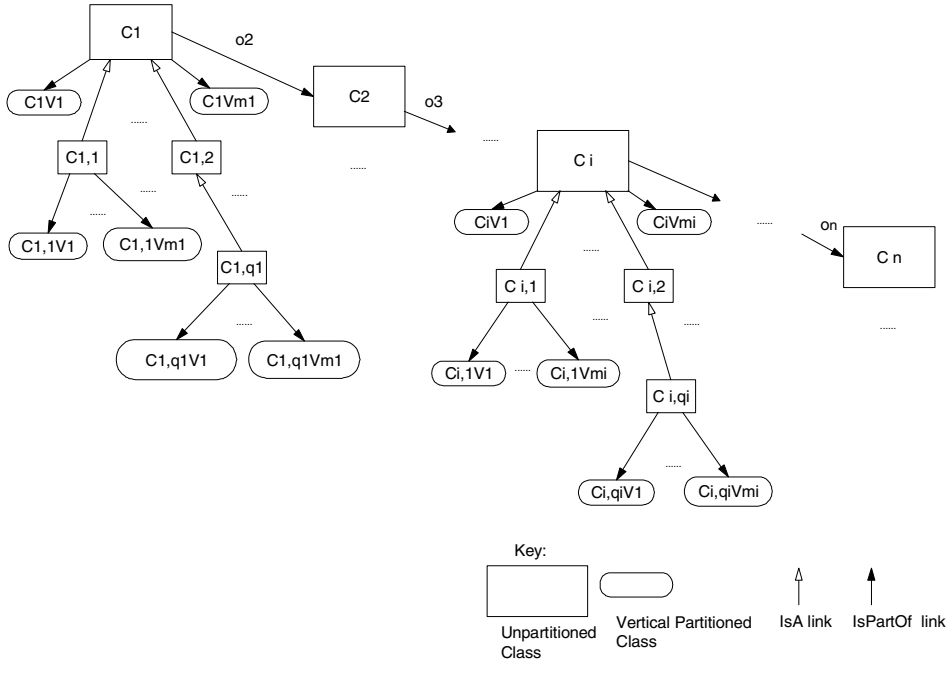


Fig. 5. Example schema for a MDG path in vertical partitioned case

predicate on the current class C). However, the Yao function only applies when $b \leq n$, i.e., when the object is smaller than or equal to the page size. For larger objects, we estimate the number of page accesses by a simple proportion: $b * k/n$. In building the cost model, we thus use an auxiliary function Y as defined below:

$$Y(n,\, b, k) = \begin{cases} Yao(n,\, b,\, k) & \text{for object size smaller than or equal to page size} \\ b * k/n & \text{for object size larger than page size} \end{cases}$$

3.2 IO cost formulae for method execution

In [11], the query processing cost along a path expression (either MDG, parameter or hidden path expression) has the following three cost components:

IOPathExpression = IOLoad + IOEval + IOBuild

where *IOLoad* is the number of disk IOs to load in the whole root class collection, *IOEval* is the total number of disk IOs to evaluate the predicate by traversing through the different classes along the path expression, and *IOBuild* is the number of disk IOs to build the results. In [8,9,10], we have extended the cost formulae of [11] to query processing in both unpartitioned and vertically partitioned OODBs.

In this paper, these three cost components are further extended to the different path expressions (namely MDG, parameter, and hidden path expressions) in a complex method. As described in Sect. 2.3, the cost of evaluating a complex method m is calculated by the general cost formula of *Cost(m)* which is dependent on *IOCost* functions of each (sub)method involved. We thus concentrate here on the cost formulae of *IOCost(m)* for a complex method m(defined in a class C_i) without regard to other method invocations. Similar to *IOPathExpression,* the *IOCost(m)* includes the following components:

IOCost(m) = IOLoad + IOMDGPath + IOBuild +

$$\sum \textit{IOHiddenPaths} + \sum \textit{IOParamPaths}$$

where: (1) *IOLoad* is the number of disk IOs to load in the whole root class collection (with C_i being the root class) of a MDG; (2) *IOMDGPath* is the number of disk IOs to evaluate "one node" in a MDG path expression which is from a previous class C_{i-1} to the current class C_i (if C_i is not the root class in the MDG)[1]; (3) *IOBuild* is the number of disk IOs to build the results; and (4) *IOHiddenPath* is the number of disk IOs to process the *whole* hidden path expression, and *IOParamPath* is the number of disk IOs to process the *whole* parameter path expression[2]. After this we use the formulae developed in Sect. 2.2.1 to calculate the total cost of invoking complex method m.

We note that sequential scan and index scan are the two major strategies used for scanning a class collection. As mentioned before, the objective of using an index is to attain faster object access, while the objective of using vertical partitioning is to reduce irrelevant data access. In the general case, since not every instance variable defined in a class has an index associated with it, the objects in the class collection are assumed to be accessed sequentially in order to have a uniform comparison of experimental results.

The cost formulae for sequential scans are summarized in Tables 3 and 4. Interested readers are referred to [8,9,10] for details. For methods that use index scan to a class collection, the cost model formulae differ only in the *IOLoad*, *IOHiddenLoad*, and *IOParamLoad* components. For example in LMH, according to [11], for non-clustered index implemented as a B^+-tree with average fan-out of b, the *IOLoad* for the unpartitioned case is

$$\log_b \left(SEL_1 * \sum_{k=0}^{q_1} \frac{\|C_{1,\,k}\|}{NP_{1,\,k}} \right) + \sum_{k=0}^{q_1} Y(\|C_{1,\,k}\|\,,\, |C_{1,\,k}|\,,\, ref_{1,\,k}),$$

and the *IOLoad* for the vertical partitioned case is

$$\log_b \left(SEL_1 * \sum_{k=0}^{q_1} \frac{\|C_{1,\,k}\|}{NP_{1,\,k}} \right) + \sum_{k=0}^{q_1} Y(\|C_{1,\,k}\|\,,\, |CO_{1,\,k}|\,,\, ref_{1,\,k}) + \sum_{k=0}^{q_1} \left[\sum_{j=1}^{m_1} M_{1,\,j,\,k} * Y(\|C_{1,\,k}\|\,,\, |C_{1,\,k}V_j|\,,\, ref_{1,\,k}) \right].$$

The *IOHiddenLoad* and *IOParamLoad* can be similarly defined.

3.3 Evaluation of vertical class partitioning

To compare and contrast the utility of vertical class partitioning, we have conducted a number of analytical experiments in [7,8,9,10]. We evaluated the effectiveness of vertical class partitioning in reducing the irrelevant data accessed for processing queries and methods. Due to the space limits, we only summarize below the findings and conclusions from these experiments:

- In [7,8,9,10], a cost model was developed for calculating the number of disk accesses required for processing a set of OQL (object query language) queries. This cost model was extended to calculate the number of disk accesses incurred for processing a set of queries and methods on a vertical class partitioned OODB.
- In [7,8,9,10], analytical experiments were conducted to show the utility of vertical class partitioning. In particular, it was shown that: (1) there is an optimal number of vertical

[1] As we are calculating the method execution cost for the current class, we only need to consider the traversal cost from the previous class towards the current class.

[2] Note that in general there may be multiple hidden / parameter path expressions within a method. *IOHiddenPath = IOHiddenLoad + IOHiddenEval* and *IOParamPath = IOParamLoad + IOParamEval*. For simplicity, our cost model concentrates on methods whose resulting values or OIDs from method execution are from the classes along the MDG path expression, not from the hidden/parameter path expressions so there is no disk IO required to build the result along the hidden/parameter path expression and hence *IOHiddenBuild* and *IOParamBuild* are all zeros.

Table 3. Cost model formulae for MI-VCP in LMH

LMH	Unpartitioned	Vertical Partitioned
IOLoad	$\sum_{k=0}^{q_1} \lvert C_{1,k}\rvert$	$\sum_{k=0}^{q_1}\left[\lvert CO_{1,k}\rvert + \sum_{j=1}^{m_1} M_{1,j,k} * \lvert C_{1,k}V_j\rvert\right]$
IOMDGPath	$\sum_{k=0}^{q_i} Y(\lVert C_{i,k}\rVert, \lvert C_{i,k}\rvert, ref_{i,k})$	$\sum_{k=0}^{q_i}[VP_i * (Y(\lVert C_{i,k}\rVert, \lvert CO_{i,k}\rvert, ref_{i,k}) + \sum_{j=1}^{m_i}(M_{i,j,k} * Y(\lVert C_{i,k}\rVert, \lvert C_{i,k}V_j\rvert, ref_{i,k}))) + (1-VP_i) * Y(\lVert C_{i,k}\rVert, \lvert C_{i,k}\rvert, ref_{i,k})]$
IOBuild	$\sum_{k=0}^{q_i} \frac{SEL_i * \lVert C_{i.k}\rVert * Sproj_{i,k}}{PS}$	$\sum_{k=0}^{q_i}\left[\sum_{j=1}^{m_i} \frac{SEL_i * \lVert C_{i.k}\rVert * Sproj_{i,j,k}}{PS}\right]$
IOHiddenLoad	$\sum_{k=0}^{hq_1} \lvert H_{1,k}\rvert$	$\sum_{k=0}^{hq_1}\left[\lvert HCO_{1,k}\rvert + \sum_{j=1}^{hm_1} HM_{1,j,k} * \lvert H_{1,k}V_j\rvert\right]$
IOHiddenEval	$\sum_{i=2}^{h}\left[\sum_{k=0}^{hq_i} Y(\lVert H_{i,k}\rVert, \lvert H_{i,k}\rvert, Href_{i,k})\right]$	$\sum_{i=2}^{h}[\sum_{k=0}^{hq_i}[HVP_i * (Y(\lVert H_{i,k}\rVert, \lvert HCO_{i,k}\rvert, Href_{i,k}) + \sum_{j=1}^{hm_i}(HM_{i,j,k} * Y(\lVert H_{i,k}\rVert, \lvert H_{i,k}V_j\rvert, Href_{i,k}))) + (1-HVP_i) * Y(\lVert H_{i,k}\rVert, \lvert H_{i,k}\rvert, Href_{i,k})]]$
IOParamLoad	$\sum_{k=0}^{pq_1} \lvert P_{1,k}\rvert$	$\sum_{k=0}^{pq_1}\left[\lvert PCO_{1,k}\rvert + \sum_{j=1}^{pm_1} PM_{1,j,k} * \lvert P_{1,k}V_j\rvert\right]$
IOParamEval	$\sum_{i=2}^{p}\left[\sum_{k=0}^{pq_i} Y(\lVert P_{i,k}\rVert, \lvert P_{i,k}\rvert, Pref_{i,k})\right]$	$\sum_{i=2}^{p}[\sum_{k=0}^{pq_i}[PVP_i * (Y(\lVert P_{i,k}\rVert, \lvert PCO_{i,k}\rvert, Pref_{i,k}) + \sum_{j=1}^{pm_i}(PM_{i,j,k} * Y(\lVert P_{i,k}\rVert, \lvert P_{i,k}V_j\rvert, Pref_{i,k}))) + (1-PVP_i) * Y(\lVert P_{i,k}\rVert, \lvert P_{i,k}\rvert, Pref_{i,k})]]$

Table 4. Cost model formulae for MI-VCP in SMH

SMH	Unpartitioned	Vertical Partitioned
IOLoad	same as LMH	same as LMH
IOMDGPath	$\sum_{k=0}^{q_i}\left[\left(\prod_{r=2}^{i} ref_{r,k}\right) * \left\lceil\frac{SC_{i,k}}{PS}\right\rceil\right]$	$\sum_{k=0}^{q_i}\left[\left(\prod_{r=2}^{i} ref_{r,k}\right) * \left(1 + \sum_{j=1}^{m_i} M_{i,j,k} * \left\lceil\frac{SC_{i,k}V_j}{PS}\right\rceil\right)\right]$
IOBuild	same as LMH	same as LMH
IOHiddenLoad	same as LMH	same as LMH
IOHiddenEval	$\sum_{i=2}^{h}\left[\sum_{k=0}^{hq_i}\left[\left(\prod_{r=2}^{i} Href_{r,k}\right) * \left\lceil\frac{SH_{i,k}}{PS}\right\rceil\right]\right]$	$\sum_{i=2}^{h}\left[\sum_{k=0}^{hq_i}\left[\left(\prod_{r=2}^{i} Href_{r,k}\right) * \left(1 + \sum_{j=1}^{hm_i} HM_{i,j,k} * \left\lceil\frac{SH_{i,k}V_j}{PS}\right\rceil\right)\right]\right]$
IOParamLoad	same as LMH	same as LMH
IOParamEval	$\sum_{i=2}^{p}\left[\sum_{k=0}^{pq_i}\left[\left(\prod_{r=2}^{i} Pref_{r,k}\right) * \left\lceil\frac{SP_{i,k}}{PS}\right\rceil\right]\right]$	$\sum_{i=2}^{p}\left[\sum_{k=0}^{pq_i}\left[\left(\prod_{r=2}^{i} Pref_{r,k}\right) * \left(1 + \sum_{j=1}^{pm_i} PM_{i,j,k} * \left\lceil\frac{SP_{i,k}V_j}{PS}\right\rceil\right)\right]\right]$

class fragments of a class to process a set of queries; (2) the proportion of benefit due to vertical class partitioning remains constant as cardinality of the relation increases, that is, vertical class partitioning is beneficial for both small and large object databases; (3) the proportionality of instance variables affects the utility of the vertical class partitioning, in that the larger the proportionality of instance variables accessed the less useful vertical class partitioning is; and (4) it is better to vertically class partition all the classes along the class composition hierarchy, especially for large fan-outs.

4 Vertical partitioning algorithms

Given an OODB schema and a set of methods with specified execution characteristics (MDGs and MID terms), vertical partitioning algorithms can be devised to generate optimal

or near-optimal partitioning schemes. The problem of creating an optimal vertical class partitioning scheme is a hard problem [26], hence heuristic algorithms are needed to be designed to come up with near-optimal vertical class partitioning schemes. Two main heuristics, namely, affinity-based [22] and cost-driven [5] have been proposed in prior research. In this section, we present two algorithms for vertical class partitioning: a cost-driven algorithm which is a pure cost-based approach, and a hill-climbing heuristic algorithm which is based on a combination of affinity-based and cost-based approaches.

4.1 Cost-driven algorithm (CDA)

The cost-driven algorithm (CDA) uses the cost model for method execution. By exhaustively searching all the partitioning schemes, it finds the optimal vertical partitioning scheme that minimizes the cost. As shown in Fig. 6, the algorithm consists of two stages. In the first stage, we perform a detailed analysis on the cost relationship and the different cost component distributions among the different classes for every method defined in the schema. In the second stage, we make use of the cost information obtained to derive the optimal vertical partitioning scheme. We begin the first stage by studying the cost relationship within the MDG for each method defined in the schema. Different types of method invocation are identified, which include simple, conditional, and repeated method invocations. (Again, we assume that the underlying OODBMS monitors the method execution and keeps track of the relative frequencies between conditional method invocations and the repeating factors under repeated invocations.) We then study the detailed cost components within each class for each method defined in the schema; these components include the IO costs involved in the different stages of method execution: *IOLoad, IOMDGPath, IOBuild, IOHiddenPath* and *IOParamPath*. We treat polymorphic/redefined methods as different methods during the analysis. At the end of the first stage, we obtain detailed cost formulae and cost distribution among the different classes for each method defined in the schema. The above information is then used in the *find_method_execution_cost_within_that_class* procedure of the CDA algorithm (see Fig. 6) to find the method execution cost. In this respect, CDA is more suitable for production OODBs (with predefined method execution characteristics) because all of the above information can be collected and stored in the system catalog.

In the second stage, we make use of the cost information to derive the optimal vertical partitioning scheme. For each class in the schema, we enumerate all possible vertical partitioning schemes of that class, and for each possible vertical partitioning scheme *Si*, we calculate the total IO required (which takes into account the cost relationship and the cost components) for each method, and sum up the grand total IO cost required for all the methods under the scheme *Si*. After enumerating all possible vertical partitioning schemes, the overall minimum cost vertical partitioning scheme for a particular class can be obtained.

```
/* first stage */
Step 1:  Perform detailed analysis on cost relationship and
         cost components
/* second stage, perform exhaustive enumeration on every class */
Step 2:  For every class in the schema do
Step 3:   min:= + ∞
Step 4:   For every possible vertical partitioning scheme in that
            class do
Step 5:     total_IO:= 0
Step 6:     For every method do
Step 7:       IOCost:= find_method_execution_cost_within_that_class
Step 8:       total_IO:= total_IO + IOCost
Step 9:     EndFor
Step 10:    If total_IO < min
Step 11:      min:= total_IO
Step 12:      optimal_configuration:= current_configuration
Step 13:    EndIf
Step 14:  EndFor
Step 15: EndFor
```

Fig. 6. Cost-driven algorithm

4.2 Hill-climbing heuristic algorithm (HCHA)

Our second algorithm combines the well-known affinity-based vertical algorithm (viz., the graph-based algorithm described in [24]) with the cost-based approach, so as to achieve a reasonable near-optimal result in an efficient manner. We start with a recap of the affinity-based algorithm first.

Pure affinity-based vertical partitioning algorithm.

As described in [24], the affinity-based algorithm starts from the instance variable affinity matrix which is generated from the instance variable usage matrix. An *instance variable usage matrix* (IVUM) represents the use of instance variables. Each row in the matrix refers to a method; a "1" entry in a column indicates that the method accesses the corresponding instance variable. An IVUM element $u_{t,i}$ is set to "1" if the tth method accesses the ith instance variable; "0" otherwise. Figure 7a shows an example IVUM.

Based on IVUM, an *instance variable affinity matrix* (IVAM) element is defined as

$$a_{i,j} = \sum_{t \in Methods} (u_{t,i}\, AND\, u_{t,j}) * freq_t \ ,$$

where the summation is over all the methods and $freq_t$ is the frequency of method execution of the tth method. Each IVAM matrix element measures the strength of an "imaginary bond" [24] between the two instance variables when they are accessed together by methods. An example IVAM (which corresponds to the above IVUM) is shown in Fig. 7b.

As in [24], the algorithm starts with the instance variable affinity matrix by considering it as a complete graph. It then forms a linearly connected spanning tree and generates all meaningful vertical fragments simultaneously by considering a cycle as a fragment. This algorithm is based on the fact that all pairs of attributes in a fragment have high intra-fragment affinity and low inter-fragment affinity. Compared with previous vertical partitioning algorithms, this algorithm has computational superiority and is of complexity $O(n^2)$ [24] (where n is the number of instance variables in the class).

		Instance variable				
		v1	v2	v3	v4	v5
method	m1	0	1	1	1	0
	m2	0	1	1	1	0
	m3	0	1	1	0	0

a

	v1	v2	v3	v4	v5
v1	0	0	0	0	0
v2	0	160	160	110	0
v3	0	160	160	110	0
v4	0	110	110	110	0
v5	0	0	0	0	0

b

Fig. 7. a Instance variable usage matrix (IVUM) and **b** instance variable affinity matrix (IVAM)

```
/* first stage */
Step 1:  Perform [24] graph-based algorithm
         /* it outputs the affinity-based optimal partitioning scheme
            and the instance variable affinity matrix for later use */
/* second stage */
Step 2:  For each class in the schema
Step 3:    curr_part_scheme:= affinity-based optimal partitioning scheme
Step 4:    old_cost:= find_cost(curr_part_scheme)
Step 5:    finished:= false
Step 6:    While not finished
Step 7:      Perform find_max_inter_fragment_affinity
Step 8:      Perform find_min_popularity
Step 9:      For next_move in [left_merge, right_merge, split_new,
                   single_split]
Step 10:       next_part_scheme:= next_move(curr_part_scheme)
Step 11:       new_cost:= find_cost(next_part_scheme)
Step 12:       If new_cost<old_cost
Step 13:         curr_part_scheme:= next_part_scheme
Step 14:         old_cost:= new_cost
Step 15:         Goto Step 19
Step 16:       EndIf
Step 17:     EndFor
Step 18:     finished:= true /* Cannot find any next state with lower cost */
Step 19:   EndWhile
Step 20: EndFor
```

Fig. 8. Hill-climbing heuristic algorithm

In our approach, we first generate the instance variable usage matrix by analyzing the method definition and then apply the graph-based algorithm [24] to generate an optimal affinity-based vertical partitioning scheme. Concerning the instance variable usage matrix, we do not need to separate the instance variables into groups (such as one group per class), as the graph-based algorithm is general enough to cater for all the instance variables in the whole schema in one shot. The algorithm of [24] is included in Appendix A for reference. Note that the affinity-based algorithm is more suitable for determining an initial vertical class partitioning scheme, as is the case in the HCHA algorithm to be described next.

The hill-climbing heuristic algorithm

As the exhaustive enumeration strategy used in the cost-driven algorithm (CDA) requires a high computational cost of order $O(n^n)$ [4] (where n is the number of instance variables in the class), it is impractical when the total number of instance variables in the schema is very large. On the other hand, the pure affinity-based approach is not as comprehensive as the cost-based approach in modeling important database characteristics, like the sizes of the instance variables. Furthermore, the affinity-based approach cannot yield an analytical cost comparison between different partitioning schemes, yet such a comparison is very crucial in comparing the effectiveness of different vertical partitioning algorithms. Some heuristic approach is therefore needed to tackle the vertical partitioning problem effectively. Here we introduce a hill-climbing heuristic algorithm (HCHA). The HCHA algorithm uses the concept of popularity which is defined by the following:

Definition. *The* popularity *of a particular instance variable v_i is the sum of the frequencies of the methods (transactions) which access v_i.*

As in [25], there are four major elements in the hill-climbing heuristics used by HCHA:

- **Initial state:** the optimal partitioning scheme generated by the graph-based algorithm of [24] is used as the initial state. This is a good choice because the affinity-based optimal partitioning scheme is closer to the real cost-based optimal partitioning scheme than any random and/or *ad hoc* initial guess.
- **Next state:** as to be explained later, we shuffle the instance variables in the different fragments in the current partitioning scheme to generate the next state. We have several "move" operations from the current state to the next state, which can involve: (1) migrating an instance variable v from one fragment *Fi* to another fragment *Fj*, so that for some instance variable w in *Fj*, (v, w) has the highest *inter-fragment affinity*; (2) grouping two instance variables from two fragments to form a new fragment; and (3) separating one or more instance variables that have the lowest popularity[3] from a fragment to form a new fragment, where the popularity for the ith instance variable v_i is defined as the ith diagonal element in the instance variable affinity matrix used by the affinity-based algorithm. The intuition is that we should group instance variables which are frequently accessed together, i.e., those instance variables having similar/comparable popularities, to form a fragment, so that the variations of the popularities within the fragment will be small.
- **Comparison:** we use the cost formulae of the cost-based approach to calculate the cost required for the next partitioning scheme, to see if it is of lower cost than the current partitioning scheme.
- **Termination:** we terminate the hill-climbing algorithm in any iteration when we cannot find any partitioning scheme (after all different ways of shuffling from the current partitioning scheme) with lower cost than the current partitioning scheme.

The HCHA algorithm as shown in Fig. 8 is thus of a two stage process: stage one is the graph-based vertical partitioning algorithm [24], and stage two is the application of the hill-climbing heuristics.

[3] Note that we use popularities to decide if a fragment of a partitioning scheme S should be split. When there is more than one

Table 5. OODB schema

Class STAFF {		Class PROJECT {	
Properties:		Properties:	
staff_name	char[40];	name	char[40];
basic_salary	int;	supervisor	STAFF;
Methods:		staff	setof STAFF;
staff_name()	char[40];	pj_loc	BRANCH;
basic_salary()	int; }	plan	char[200];
		Methods:	
		name()	char[40];
		supervisor()	STAFF;
		staff()	setof STAFF;
		pj_loc()	BRANCH;
		plan()	char[200];
		expense()	int; }
Class ADM_STAFF IsA STAFF {		Class BRANCH {	
Properties:		Properties:	
fringe_benefit	int;	br_name	char[40];
Methods:		estate_tax	int;
fringe_benefit()	int;	br_locs	setof SITE;
income()	int; }	br_staff	setof STAFF;
		function	char[100];
		Methods:	
		br_name()	char[40];
		estate_tax()	int;
		br_locs()	setof SITE;
		br_staff()	setof STAFF;
		function()	char[100];
		br_cost()	int; }
Class TECH_STAFF IsA STAFF{		Class SITE {	
Properties:		Properties:	
ot_hours	int;	address	char[120];
ot_hourly_rate	int;	rent_per_sqft	int;
Methods:		size	int;
ot_hours()	int;	district	char[20];
ot_hourly_rate()	int;	tel	char[8];
income()	int; }	Methods:	
		address()	char[120];
		rent_per_sqft()	int;
		size()	int;
		district()	char[20];
		tel()	char[8];
		rent()	int; }

The procedure *find_cost* calculates the total cost of method execution of the input partitioning scheme using the cost-based formulae. There are four procedures in the algorithm to find the "next move": (1) *left_merge,* (2) *right_merge,* (3) *split_new* , and (4) *single_split*. Given the current partitioning scheme (*curr_part_scheme*), procedure *find_max_inter_fragment_affinity* finds a pair of instance variables from two different fragments with maximum inter-fragment affinity (by consulting the instance variable affinity matrix produced in stage one). The three procedures *left_merge*, *right_merge* and *split_new* all require the maximum inter-fragment affinity pair. For example, if instance variables B and D are of maximum inter-fragment affinity for the current partitioning scheme: $\{$*(A B C) (D E) (F G)*$\}$, then procedures *left_merge, right_merge* and *split_new* will yield $\{$*(A B C D) (E) (F G)*$\}$, $\{$*(A C) (B D E) (F G)*$\}$ and $\{$*(A C) (B D) (E) (F G)*$\}$, respectively. On the other hand, procedure *find_min_popularity* finds within the fragment the instance variable of minimum popularity but of the largest variation in popularity. For example, if <(100 60 80) (0 0) (50 50)> is the popularity for a current partitioning scheme $\{$*(A B C) (D E) (F G)*$\}$, the minimum popularity instance variable will then be B since (ABC)has larger variation than(FG). Therefore

fragment in S, we select the fragment F with the largest variation among the popularities of its instance variables.

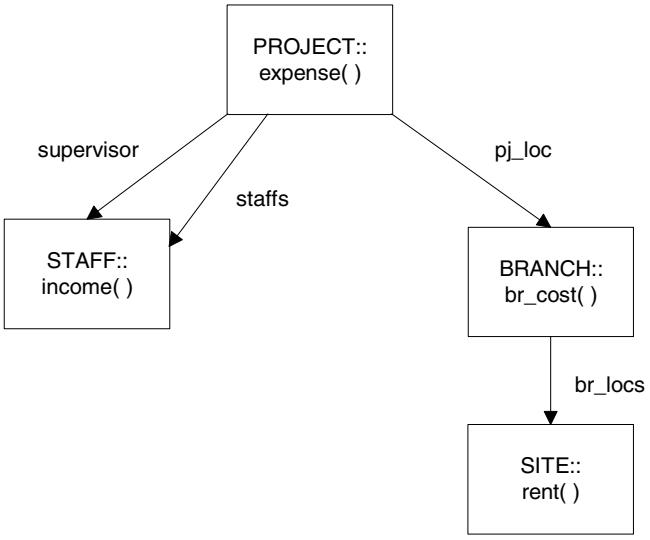


Fig. 9. MDG of complex method *expense()*

the next partitioning scheme will become $\{$*(A C) (D E) (F G) (B)*$\}$ after applying the procedure *single_split*.

Note that HCHA will not enter an infinite loop or fail to terminate. This is because: (1) the total number of different partitioning schemes is finite; and (2) our *old_cost* is ever decreasing, so if the algorithm generates a particular partitioning scheme that has been generated before, this partitioning scheme will not be considered as the next one, since the cost of it cannot be less than itself. Another characteristic of this algorithm is that it either obtains an improved partitioning scheme or terminates at once. As our initial partitioning scheme is the affinity-based optimal partitioning scheme generated from the [24] algorithm, HCHA is guaranteed to always produce a partitioning scheme which is at least as good as that of the [24] algorithm.

5 Evaluation of the vertical partitioning algorithms

In this section we present the results of analytical evaluation experiments on the vertical partitioning algorithms (viz., CDA and HCHA) introduced in the previous section. For better comparing and contrasting purposes, we treat here the pure affinity-based algorithm [24] as a separate, additional algorithm for vertical class partitioning. Therefore, the pros and cons of the pure affinity algorithm are also discussed along with that of CDA and HCHA.

According to the result of actual experimental studies we have conducted (see Sect. 6.3), LMH is a good approximation to the actual method execution cost for large objects, therefore we shall concentrate on the LMH case in all our subsequent experiments.

5.1 OODB schema

We use the OODB schema as shown in Table 5 for our analytical evaluation experiments.

Case (i) – even distribution of instance variable accesses:

Case (i) Class	# of iterations	# of partitioning schemes checked	IO cost of initial partitioning scheme by HCHA	IO cost of final partitioning scheme by HCHA	Optimal IO cost as predicted by CDA
PROJECT	6	17	14920	9520	9520
STAFF	2	8	27790	27740	27740
BRANCH	6	19	13590	9300	9300
SITE	5	15	22830	17440	17440

5.2 Example methods

In our example, the method definitions are divided into two categories: **(1) elementary instance variable access (EIVA) methods**; and **(2) application methods.** EIVA methods are defined on a one-to-one correspondence with the instance variables of a class. They are used to access the instance variable values, which facilitates the encapsulation properties of the OODB model and the hiding of the implementation details of the instance variables. For example, method *estate_tax()* returns the estate tax of the *BRANCH* class object. Application methods are defined by the application designer to access the database. Usually these methods access multiple instance variables, and may return some results and/or invoke other methods. In our example, we have the following methods belonging to this category:

- *income()* in class *TECH_STAFF* returns the salary as calculated by *basic_salary* + *ot_hours* * *ot_hourly_rate*;
- *income()* in class *ADM_STAFF* returns the salary as calculated by *basic_salary* + *fringe_benefit*; (Note that *TECH_STAFF* and *ADM_STAFF* are subclasses of *STAFF*, and they have different definitions for the *income()* method.)
- *rent()* in class *SITE* returns the rent as calculated by *rent_per_sqft* * *size*;
- *br_cost()* in class *BRANCH* returns the cost as calculated by adding *estate_tax* with the sum of rents of all *br_locs* sites;
- *expense()* in class *PROJECT* returns the expense as calculated by adding salary of supervisor with the sum of salary of all staff and the *br_cost* of *pj_loc* branch. The MDG of this complex method is illustrated in Fig. 9.

5.3 Method execution environment

Table 6 presents the method execution environment for this analytical evaluation. To better understand the query processing requirements of the method execution environment, the methods are expressed in the form of Object Query Language (OQL). There are eight queries: $q1$ to $q8$, that correspond to eight methods: $m1(\ )$ to $m8(\ )$. Table 7 presents the MDGs of the eight methods. In the last column of Table 7, the relative method execution frequencies are shown. For method execution cost calculation, the eight methods in the MDG (as shown in Fig. 10) can be viewed as sub-methods of a dummy method m. The dummy method contains no method execution code and hence requires no disk access for method execution. This dummy method m is created to facilitate the formulation of method execution cost. The total method execution cost is given by:

Total Method Execution Cost

$= \dfrac{f_1 * Cost(m1) + \ldots + f_8 * Cost(m8)}{f_1 + \ldots + f_8}$. That means the total method execution cost is the weighted sum by frequency of the individual methods' execution costs.

5.4 Results

Our analytical evaluations are performed using the cost model of Sect. 3 on the three vertical partitioning algorithms for two different cases:

- Case (i) – This corresponds to the case of rather even distribution of instance variable accesses[4], meaning that the expected performance gain of using vertical partitioning will not be substantial. The method execution environment is as shown in Tables 6 and 7. In Fig. 11, we show the Method Usage Matrix indicating the EIVA methods that are accessed by the methods $m1(\ )$ to $m8(\ )$.
- Case (ii) – This is the case of skewed instance variable accesses[5], meaning that some instance variables are heavily accessed and the expected performance gain of using vertical partitioning will be substantial. In Fig. 12, we show the Method Usage Matrix for this case.

5.4.1 The HCHA approach

We first present the results on the HCHA approach. To be concise, the results for the STAFF class actually represent the results for the whole STAFF class inheritance hierarchy.

As shown in the above table, all four classes' costs based on the HCHA approach attain the optimal IO costs as predicted by the pure cost-based approach. The conclusion is that HCHA's predicting power of the optimal partitioning scheme is very good. In addition, note that the total number of partitioning schemes checked in the CDA approach using the exhaustive enumeration strategy is 52 for each of the four classes[6], but the number of partitioning schemes checked in the HCHA case

[4] Even distribution refers that every instance variable is accessed by the methods.

[5] Skewed instance variable accesses means that some instance variables are heavily accessed, and some instance variables may never be accessed by the methods. The instance variables that are never accessed by the methods in this case are: plan (in class PROJECT), staff_name (in class STAFF), br_staffs, function (in class BRANCH), and address, tel (in class SITE).

[6] For a class with five instance variables (say A, B, C, D and E), the different partitioning schemes include a one-fragment partitioning scheme – {(A B C D E)}, 15 two-fragments partitioning schemes

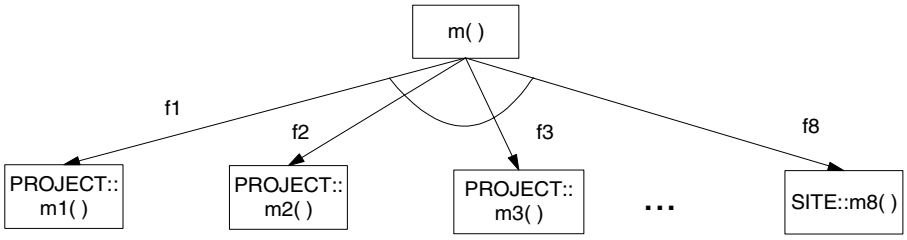


Fig. 10. MDG of method execution environment

Table 6. Method execution environment

Query	OQL	Method
$q1$	select p.name from p in PROJECT where expense(p) > 2000000;	$m1( )$
$q2$	select p.name, s.basic_salary from p in PROJECT, s in TECH_STAFF where s in p.staffs and income(s) > 10000;	$m2( )$
$q3$	select p.name, s.staff_name from p in PROJECT, s in ADM_STAFF where s=p.supervisor and s.basic_salary > 50000;	$m3( )$
$q4$	select b.br_name, st.address, st.district, st.tel from p in PROJECT, b in BRANCH, st in SITE where b in p.pj_loc, st in b.br_locs;	$m4( )$
$q5$	select b.br_name, b.function, st.size, st.district from b in BRANCH, st in SITE where st in b.br_locs;	$m5( )$
$q6$	select p.staffs, p.plan from p in PROJCET where p.name="OODB";	$m6( )$
$q7$	select b.br_name, b.function from b in BRANCH;	$m7( )$
$q8$	select st.address, st.tel from st in SITE where st.district="HK";	$m8( )$

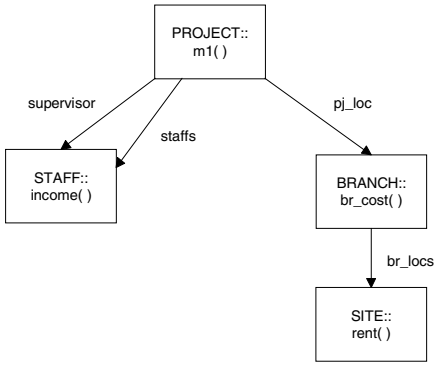
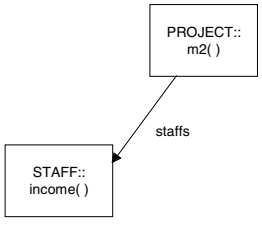
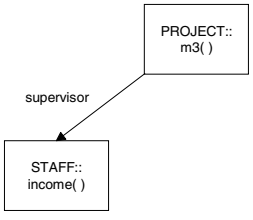
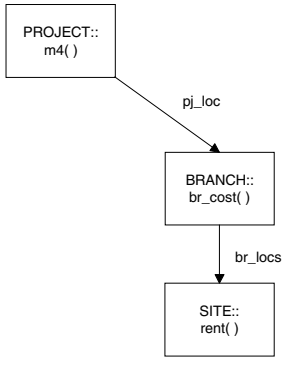
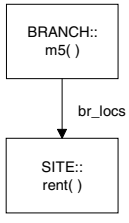
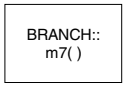
Method	name( )	supervisor( )	staffs( )	pj_locs( )	plan( )	staff_name( )	basic_salary( )	fringe_benefit( )	ot_hours( )	ot_hourly_rate( )	br_name( )	estate_tax( )	br_locs( )	br_staffs( )	function( )	address( )	rent_per_sqft( )	size( )	district( )	tel( )	frequency
m1( )	√	√	√	√			√	√	√	√		√	√				√	√			100
m2( )	√		√			√	√		√	√											40
m3( )	√	√				√	√														50
m4( )				√		√					√		√	√		√			√	√	50
m5( )											√		√		√			√	√		50
m6( )	√		√		√																10
m7( )											√				√						20
m8( )																√			√	√	30

Fig. 11. Method usage matrix for case (i)

Method	name( )	supervisor( )	staffs( )	pj_locs( )	plan( )	staff_name( )	basic_salary( )	fringe_benefit( )	ot_hours( )	ot_hourly_rate( )	br_name( )	estate_tax( )	br_locs( )	br_staffs( )	function( )	address( )	rent_per_sqft( )	size( )	district( )	tel( )	frequency
m1( )	√	√	√	√			√	√	√	√		√	√				√	√			20
m2( )	√		√				√		√	√											20
m3( )	√	√					√														20
m4( )											√		√				√	√	√		20
m5( )							√														40
m6( )	√	√	√	√																	40
m7( )											√		√								40
m8( )																		√	√		40

Fig. 12. Method usage matrix for case (ii)

Table 7. MDGs of method execution environment

Method	MDG	Frequency
*m*1( )		100
*m*2( )		40
*m*3( )		50
*m*4( )		50
*m*5( )		50
*m*6( )		10
*m*7( )		20
*m*8( )		30

Case (ii) – skewed distribution of instance variable accesses:

Case(ii) Class	# of iterations	# of partitioning schemes checked	IO cost of initial partitioning scheme by HCHA	IO cost of final partitioning scheme by HCHA	Optimal IO cost as predicted by CDA
PROJECT	5	13	5280	4600	4520
STAFF	2	5	2560	2500	2500
BRANCH	4	10	2080	1820	1820
SITE	4	10	2700	2280	2280

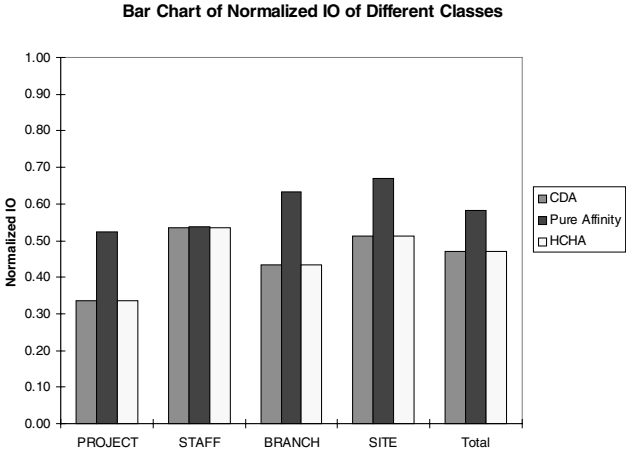


Fig. 13. Normalized IO for case (i) of even distribution of instance variable accesses

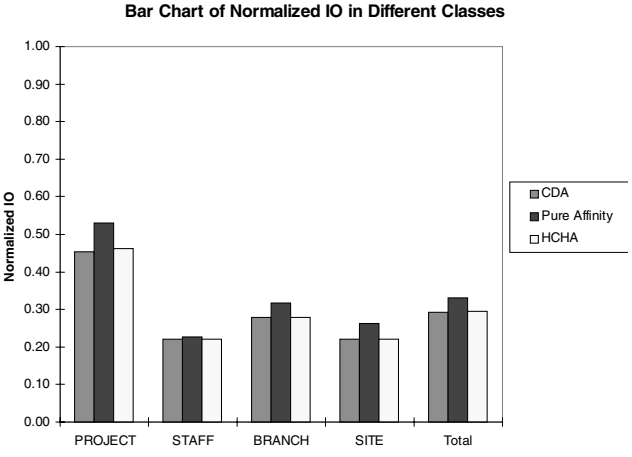


Fig. 14. Normalized IO for case (ii) of skewed distribution of instance variable accesses

is much lower than the CDA approach. Thus, a companion conclusion is that the HCHA approach is more efficient than the CDA approach in the even distribution case.

In this case, the distribution of instance variable accesses is skewed, meaning that some of the instance variables are heavily accessed and some of the instance variables are never accessed by the methods[7]. As shown in the above table, the HCHA approach attains the optimal IO costs for three out of the four classes as predicted by the CDA approach. For the missed case (i.e., *PROJECT* class), the IO cost of the final partitioning scheme is 4,600, which means only a 2% error (since $4,600/4,520 = 1.02$). The conclusion is that HCHA's predicting power of the optimal partitioning scheme is still good. In addition, we can see that the number of partitioning schemes checked in this case is much lower than the CDA's exhaustive enumeration strategy (which requires 52). This again confirms that HCHA is much more efficient than the CDA even for the skewed case.

– {(A)(B C D E)}, {(A B)(C D E)}, ..., etc., 25 three-fragments partitioning schemes – {(A)(B)(C D E)}, {(A)(B C)(D E)}, ..., etc., 10 four-fragments partitioning schemes – {(A)(B)(C)(D E)}, {(A)(B)(D)(C E)}, ..., etc., and one five-fragments partitioning scheme – {(A)(B)(C)(D)(E)}. Thus the total number of partitioning schemes is 52.

[7] Our definition of skewness is based on the total number of instance variables accessed by all methods. Another interesting definition would be based on a per method basis.

5.4.2 Comparisons

The results of all the three algorithms are now summarized and compared as follows:

Computation cost required by the algorithms: as CDA uses an exhaustive enumeration strategy, the computation cost is very high for a large number of instance variables. For the pure affinity-based approach, the graph-based algorithm is very efficient and the computation cost of the algorithm is low. The computation cost for the HCHA is always between the CDA and the pure affinity-based approach; in addition, the performance is dependent upon the instance variable access patterns: if the initial guess from the pure affinity-based approach is already close to optimal, then the extra computation cost to obtain the optimal scheme will be very low.

IO cost gain: the performance metric is the Normalized IO, which is defined as the cost ratio between vertical partitioned class and unpartitioned class. In the case of even distribution of instance variable accesses, Fig. 13 illustrates that the Normalized IO (in the Total column) of the HCHA approach has the same performance as that of the CDA approach, which is $1.0 - 0.47 = 53\%$ better than the unpartitioned case. On the other hand, the pure affinity-based approach is $1.0 - 0.58 = 42\%$ better than the unpartitioned case. Therefore, in the case of even distribution of instance variable accesses, the CDA and HCHA approaches are the best and they are $(0.53 - 0.42)/0.53 = 21\%$ better than the pure affinity-based approach. For skewed distribution of instance variable accesses, Fig. 14 shows that the Normalized IO (in the Total column) of the HCHA approach has similar performance as CDA, namely they are $1.0 - 0.29 = 71\%$ better than the unpartitioned case (which is a quite substantial gain); the pure

affinity-based approach is $1.0 - 0.33 = 67\%$ better than the unpartitioned case. Therefore, in the case of skewed instance variable accesses, the CDA and HCHA approaches are still the best, even though they are only $(0.71 - 0.67)/0.71 = 6\%$ better than the pure affinity-based approach in this skewed case. We note that as the instance variable accesses become more skewed, the differences in the ability to identify the optimal partitioning scheme for the different approaches become smaller.

Comparing the pros and cons of the three algorithms:

(1) Cost-driven algorithm:

(a) this has the advantage of obtaining the optimal partitioning scheme;
(b) it has a high computation cost if the number of instance variables is large;
(c) it is most suitable for a production OODB system.

(2) Pure affinity-based algorithm:

(a) it has the advantage of low computation cost even with a large number of instance variables;
(b) it has the disadvantage that it only considers very limited (transaction) characteristics, and does not take into consideration other important database characteristics such as the size of instance variables. For example, when we change the size of the instance variable name in the PROJECT class from 40 bytes to 10 bytes, the cost-driven approach would respond to the change and produce a new optimal partitioning scheme. However, as the affinity-based approach does not consider the sizes of the instance variables, it does not adapt to the change and would retain the old partitioning scheme;
(c) it is more suitable for determining an initial vertical class partitioning scheme.

(3) HCHA:

(a) it has a comparable predicting power as CDA;
(b) it uses the cost-based formulae to compare for optimality, thus can respond to database characteristic changes (such as the change in instance variable sizes);
(c) it avoids the exhaustive enumeration of all the possible partitioning schemes, hence the computation cost is more acceptable for a large number of instance variables;
(d) as with other hill-climbing algorithms, there is a chance of not finding the globally optimal solution;
(e) it is most suitable both for determining an initial vertical class partitioning scheme and for a production OODB system.

6 Empirical validation of cost model and discussions

The cost model we have developed (see Tables 3 and 4) has two distinct parts: one for the unpartitioned case and the other part for the vertically partitioned case. As our unpartitioned cost model stems from [11] in which the cost model for query execution for unpartitioned case has been validated, we concentrate on validating the cost model in the vertical partitioned scenario. In this section, we first report the evaluation on the utility of vertical partitioning by reporting the performance gain obtained from vertically partitioning a single class; we then validate our cost model by using a more general (complex) schema. We confirm that the experimental number of disk IO required for method execution is bounded by the theoretical SMH (upper bound) and LMH (lower bound) calculations. The experimentation is done on NeoAccess System – an OODB tool kit [23]. When validating the cost model, we use the *Unix time utility* to count the actual number of physical disk accesses in order to eliminate the effect of memory contention.

6.1 Experiment one: validating the utility of vertical partitioning in a single class

The aim of this experiment is to testify that vertical partitioning can indeed yield performance gain. We have implemented vertical partitioning of a single class on NeoAccess, which allowed us to study the effect of object size on the performance gain flexibly.

The implementation has the following schema:

```
Class Emp {
Properties:
  DeptInfo          Dept;
  EmpId             char[8*P];
  EName             char[50*P];
  Skill             char[200*P];
  EAddress          char[254*P-10];
Methods:
  DeptInfo()        Dept;
  EmpId()           char[8*P];
  EName()           char[50*P];
  Skill()           char[200*P];
  EAddress()        char[254*P-10];
}
```

In the NeoAccess System, each object has a unique object identifier (OID) of 6 bytes in length, and each reference pointer to another object instance (e.g., *DeptInfo*) uses 4 bytes. The parameter "P" in the above schema is a multiplication factor so that we can vary the size of the object (e.g., if we set $P = 2$, then the size of *Emp* object will be 1,024 bytes = 1 kB.)

Vertical partitioning scheme

We vertically partitioned the class *Emp* into four fragments: fragment 1-(*DeptInfo*), fragment 2-(*EmpId*, *EName*), fragment 3- (*Skill*), and fragment 4-(*EAddress*). These four fragments thus serve as "component objects" of a composite object. The composite object contains its own OID and the four object references to these fragments. The cardinality of the class *Emp* is 1,000 and the page size is 8 kB.

In table on top of the next page, the notation of "ME" on a method row means that the method uses an elementary *instance variable access method* (EIVAM) for method execution; the notation of "RL" means method uses that EIVAM for building up of the result; "frequency" is the relative conditional method execution frequency and "selectivity" is the selectivity of method m_i. Figure 15 shows the MDG for the

Method execution environment

The method execution environment is as follows:

Method	DeptInfo()	EmpId()	EName()	Skill()	EAddress()	Frequency	Selectivity
$m1$		ME	ME,RL	ME,RL		100	0.05
$m2$		ME,RL	ME,RL	ME		10	0.5
$m3$		ME,RL	ME			50	0.1

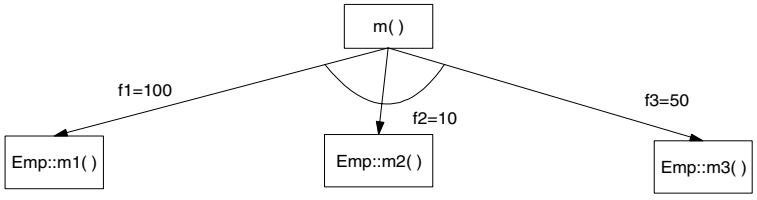


Fig. 15. MDG of method execution environment for validating the utility of vertical partitioning in a single class

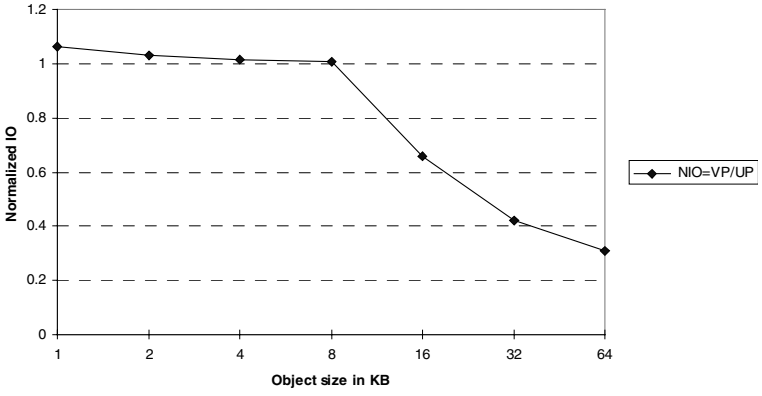


Fig. 16. The plot of NIO vs object size to see the effect on performance gain variation by object size in single class

method execution environment under discussion. The cost of method execution is given by:

$$\begin{aligned} &Cost(m) \\ &= IOCost(m) + \frac{f_1}{f_1+f_2+f_3}Cost(m1) \\ &\quad + \frac{f_2}{f_1+f_2+f_3}Cost(m2) + \frac{f_3}{f_1+f_2+f_3}Cost(m3) \end{aligned}$$

Two result plots from this experiment are enclosed here (see Figs. 16 and 17), one with larger range in the object size in order to show the general trend, and the other plot with smaller range in the object size (close to the page size of the system) so as to observe the performance gain more closely. The improvement of performance is characterized by the Normalized IO metric:

$$\text{Normalized IO} = \frac{\text{Number of disk IOs for vertical partitioned class(es) case}}{\text{Number of disk IOs for unpartitioned class(es) case}}$$

If the value of Normalized IO (NIO)is less than 1.0, it implies vertical partitioning is beneficial.

Observations

Figure 16 shows the plot of NIO vs object size. The plot naturally breaks into two different regions, with the break point at object size around 8 kB. We observe that when the object size is less than 8 kB, the NIOs are close to, but still above, 1.0, meaning that the performance of the vertical partitioned case is only comparable to that of the unpartitioned case and that the vertical partitioned case requires a few more disk IOs than the unpartitioned case due to extra overhead. However, for the region with object sizes ranging from 16 kB to 64 kB, the NIO decreases from 0.66 to 0.31. For the 64 kB case, the saving in disk IO is $1.0 - 0.31 = 69\%$ which is quite substantial. When the object size increases from 16 kB, the NIO decreases; when the object size further increases, the NIO decreases (but at a slower rate and tends to flatten for very large object size).

In order to find the critical object size that has better performance than unpartitioned case, we plot in Fig. 17 with more data points close to the 8 kB region. Concluding from the figure, when the object size is 9 kB or larger, the vertical partitioned case performs better than their unpartitioned counterpart. Knowing that the page size of our experiment is 8 kB, the observation is that the page size plus a fixed amount is the critical size which makes vertical partitioning more effective. This is because, when implementing vertical partitioning, we have an extra operating system overhead for maintaining quite a number of very small composite objects that store the reference pointers to different vertical fragments. When the size of the unpartitioned object is larger than the critical size, it becomes more beneficial, as expected.

6.2 Experiment two: validation of cost model

Our cost model has important assumptions concerning the availability of free memory buffers for objects. For the LMH

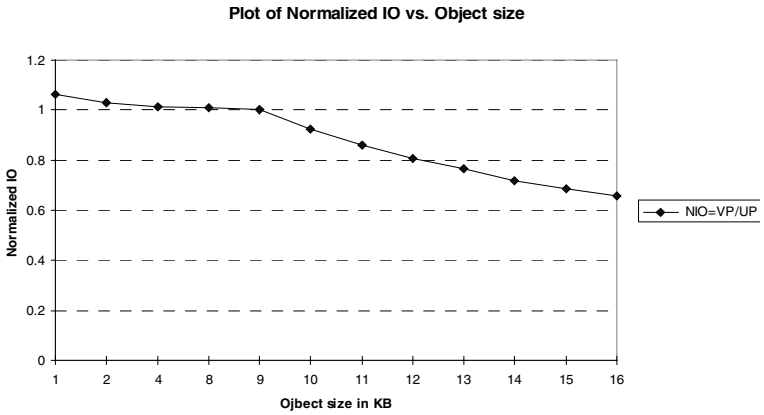


Fig. 17. The plot of NIO vs object size to observe the break point for performance gain

Class Emp {		Class Dept {		Class Proj {	
Properties:		Properties:		Properties:	
DeptInfo	Dept;	ProjInfo	setof Proj;	ProjType	char[8];
EmpId	char[8];	DeptId	char[8];	PId	char[8];
EName	char[50];	DName	char[50];	PName	char[50];
Skill	char[200];	DeptType	char[10];	Priority	char[2];
EAddress	char[100];	DAddress	char[100];	Location	char[100];
Methods:		Methods:		Methods:	
DeptInfo()	Dept;	ProjInfo()	setof Proj;	ProjType()	char[8];
EmpId()	char[8];	DeptId()	char[8];	PId()	char[8];
EName()	char[50];	DName()	char[50];	PName()	char[50];
Skill()	char[200];	DeptType()	char[10];	Priority()	char[2];
EAddress()	char[100];	DAddress()	char[100];	Location()	char[100];
}		}		}	

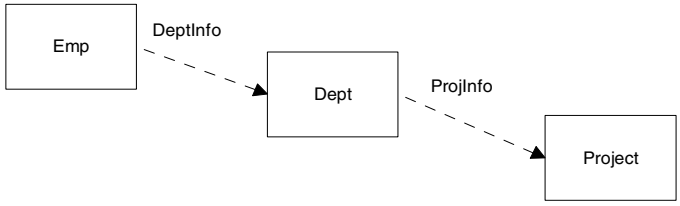


Fig. 18. Example schema for experiment two

case, we have very large memory page buffers for the objects so that no object needs to be retrieved twice. This corresponds to an ideal case for the cost model, i.e., the disk IO predicted by LMH will be the lower bound to the actual vertical partitioned implementation. For the SMH case, we have the other extreme that we only have one memory page buffer for every class/class fragment. This means when traversing the class composition hierarchy to evaluate the predicate, the same object may be required to be loaded in multiple times, causing a high increase in the number of disk IOs required. This corresponds to the worst-case scenario for the cost model, i.e., the disk IO cost predicted by SMH will be the upper bound to the actual vertical partitioned implementation. In this second experiment, our aim is to demonstrate that the actual implementation disk IO costs are bounded by the SMH and LMH theoretical calculations of our cost model, thereby confirming that our cost model is very general and can support quite a number of database characteristics.

Implementation setup

We start with an elementary method execution environment which has only MDG paths. As hidden and parameter paths have a similar cost formula as the MDG paths, once the validity of an MDG path is confirmed, so are the validities of the cost model for hidden and parameter paths. Because the subclass hierarchy is modeled by our cost model as a sum of cost factors over all the subclasses, we thus concentrate here on the class composition hierarchy. The schema for this experiment is shown in Fig. 18, with the detailed descriptions of each class given below. Similar to experiment one, we also have a P factor. For example, a P factor of 10 means that all the sizes of value-based instance variables are scaled up by 10. However, the sizes of object-based instance variables (OBIVs) are not scaled up since OBIVs are pointer references and their sizes are fixed.

In this experiment, we use the following vertical partitioning schemes which are obtained from our previous theoretical calculation [8] upon the same database schema, method characteristics, and frequencies: (1) Class *Emp* has four fragments: fragment 1-(*DeptInfo*), fragment 2-(*EmpId*, *EName*), fragment 3-(*Skill*), and fragment 4-(*EAddress*); (2) Class *Dept* has three fragments: fragment 1- (*ProjInfo*, *DeptId*), fragment 2-(*DName*, *DeptType*), and fragment 3-(*DAddress*); and (3) Class *Proj* also has three fragments: fragment 1-(*ProjType*, *PId*), fragment 2-(*PName*), and fragment 3-(*Priority*, *Loca-*

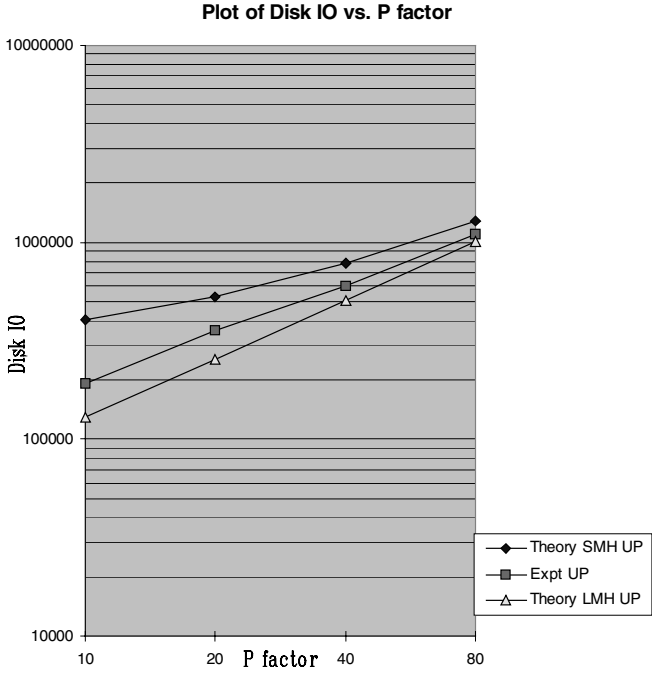


Fig. 19. Plot of disk IO vs object size for unpartitioned case

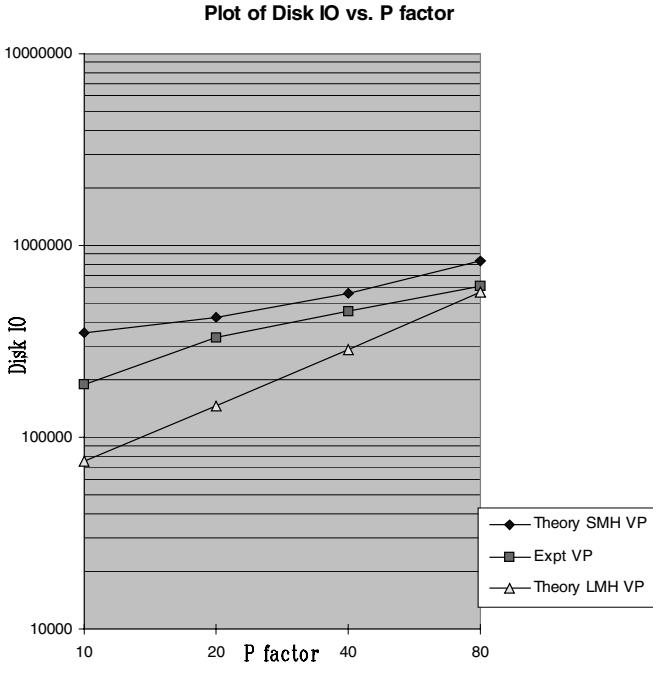


Fig. 20. Plot of disk IO vs object size for vertically partitioned case

tion). The cardinalities for the classes are as follows: *Emp* is 1,000, *Dept* is 200 and *Proj* is 1,000.

Results and observations

Due to the space limits, we only present results on the *Emp* class here; the results of the other two classes follow the same trend. Figure 19 shows the plot of disk IO vs the object size (for the unpartitioned *Emp* class). The figure shows three curves: "Theory SMH UP" – the theoretical calculation result from the SMH cost model in the unpartitioned case, "Expt UP" – the actual experimental result from the NeoAccess System in the unpartitioned case, and "Theory LMH UP" – the theoretical calculation result from the LMH cost model in the unpartitioned case. We observe that the experimental result curve is actually bounded by the SMH (as upper bound) and LMH (as the lower bound) curves. This verifies that our cost model is sound and realistic. Furthermore, as the object size increases, the "Expt UP" curve approaches the "Theory LMH UP" curve, which means for large objects (of the range sizes in our experiment), the LMH cost formula is a good approximation to the experimental results. We observe that the *Emp* class object size ranges from 3.6 kB to 28.8 kB, the SMH buffer size is just the size of one object and the LMH buffer size is the total size for all (1,000) *Emp* class objects. The LMH buffer size is more comparable to the main memory size (16 MB), and hence LMH is a more reasonable approximation to the experimental results. The SMH buffer size is too small when compared with the main memory size, and hence represents a too pessimistic or too high estimation of the actual number of disk IOs. That is, as the size of the object increases, the disparity between the amount of main memory required to store the objects and the amount of main memory allocated for the objects by the NeoAccess system reduces. Hence, for large objects, LMH is a valid assumption.

Figure 20 shows a similar plot of disk IO vs the object size (for the unpartitioned *Emp* class) in the vertically partitioned case. Again, the legend "Theory SMH VP" – denotes theoretical calculation results from the SMH cost model in the vertical partitioned case, and the other legends have similar meanings to those of Fig. 19. In Fig. 20, we also observe that the experimental curve is bounded by the SMH upper bound and LMH lower bound, which confirms that our cost model for vertical partitioned case is also sound and realistic. Similar to the unpartitioned case, for large object (of the range of object sizes in our experiment) the LMH cost formula is a good approximation to the experimental results.

6.3 Summary of experimental results

The conclusions from these two experiments are summarized as follows: (1) vertical partitioning can effectively reduce disk accesses for method execution; (2) theoretical calculations based on our cost model of SMH and LMH are realistic bounds for actual disk IO costs for method executions; (3) vertical partitioning favors large-sized object; and (4) for large objects (of the range sizes in our experiment), LMH cost formulae are good approximation to the estimation of the actual method execution costs.

6.4 Discussion

- The aim of vertical class partitioning (VCP) is to reduce the amount of irrelevant data accessed by an application [17,18,19]. This is especially useful for the next generation database applications, such as document management, multimedia, and hypermedia systems, in which many of the instance variables tend to be very large objects that should not be accessed if they are not actually needed by

the applications. Although in this paper we illustrate VCP using a centralized OODB environment, by extending the cost model and algorithm, it is also applicable to a distributed OODB environment.

- Although we illustrate the VCP in OODBs, it is equally applicable for object-relational databases, as both OODBs and object-relational databases share the same object-oriented technology.
- The analytical results (see Sect. 5.4) show that VCP algorithms require judicious incorporation based on the database system parameters, database characteristics, and application characteristics. Thus, if these characteristics cannot be accurately determined, then the applicability of the proposed techniques is diminished. However, on the whole, the cost model provides general guidelines to compare the utility of these algorithms to different database and application characteristics.
- Given the parameter values of a database application, with the help of the cost model, we can use the HCHA to set up the initial VCP scheme. When the database and/or the query processing environment change, there may be a need to evolve the VCP scheme and to reorganize the vertical class fragments. Our viewpoint is that, as we are using a cost-based approach, we are in a better position than an affinity-based approach. We can use the cost model to estimate the costs of reorganizing towards a new VCP scheme, or to estimate the extra processing cost if we continue to use the current VCP scheme. As the affinity-based approach does not guarantee generating the cost-optimal VCP scheme, it and hence will not be too useful in predicting the cost for reorganization or the extra processing cost if we continue using the current VCP scheme. From the above discussion, we conclude that VCP is more suitable to be applied to a production type OODB system[8] with a rather static database and query processing characteristics. For an evolving type of OODB system[9] with a highly dynamic database and query processing characteristics, VCP can still be applied, but we need extra processing to monitor the need for reorganizing and to perform the reorganization of the existing VCP scheme.
- Support for VCP in current commercial OODBs is still in its infancy. However, with the results showing the utility of VCP [8,10,9], it is promising that VCP can be well supported by OODB systems in the near future. Even if this is not the case, for specific applications, VCP can be hard-coded to provide efficient execution of the applications.

As a cost model for query processing is an important component in the optimizer of an Object-Oriented Databases Management System (OODBMS), and we are using a cost-based approach in the HCHA, we are in a better position to incorporate the HCHA into query processing in an OODBMS than the affinity-based approach.

[8] A production type of OODB system is a system with predefined query processing requirements. Such a system is usually used in daily operations of an organization.

[9] An evolving type of OODB system is a system with evolving/dynamic query processing requirements. Such a system is usually used in handling ad hoc requests in the organization.

7 Conclusions

Vertical partitioning in object-oriented databases (OODBs) is a challenging but very effective and relevant problem. Although a similar problem has been addressed in relational database systems, the complexity of OODB models involving subclass hierarchy and class composition hierarchy complicates the problem and thus requires new approaches to be developed. Due to the encapsulation property of OODBs, methods govern the nature of optimal (or near optimal) vertical partitioning schemes. In our earlier work [8,10,9], we have already shown the utility and effectiveness of vertical partitioning in OODB systems. In this paper, we further took into consideration the methods being invoked during OODB processing to generate an optimal/near-optimal vertical partitioning scheme. Specifically, we have developed, based on MDGs (method dependency graphs), a general-purpose cost model for method execution, and applied this model to yield vertical partitioning algorithms. Two algorithms have been developed to exploit method execution for deriving optimal/near-optimal vertical partitioning schemes. The first algorithm is the cost-driven algorithm which uses the cost model for method execution to exhaustively search all the partitioning schemes, so as to find the optimal vertical partitioning scheme. This algorithm is useful for comparing the effectiveness of the other algorithms, but may not be practical since it has a high cost of $O(n^n)$ where n is the number of instance variables. The second algorithm, HCHA, applies an affinity-based algorithm [24] to obtain an initial partitioning scheme, then applies hill-climbing heuristics to improve this solution by using the cost model for method execution. It is shown that the HCHA (which is significantly more efficient than the cost-based approach) generates an optimal or near optimal scheme. The HCHA approach thus represents a good compromise, and it can generate a solution that is at least as good as the solution provided by the affinity-based approach; in most cases, it will yield the optimal partitioning scheme.

With the help of method transformation, our method-induced VCP technique can easily support fragmentation transparency [18, 19]. This fragmentation transparency support places our method-induced VCP technique in a better position to be incorporated into an OODBMS than the affinity-based approach. On the other hand, in order to obtain an effective VCP scheme using our method-induced VCP technique, we need to obtain high-quality estimations of a number of method execution parameters. This can be an obstacle if the underlying OODBMS does not provide adequate estimations on the various method execution parameters.

Further research issues concerning this work include the application of our VCP technique to the problem of query optimization, and to the development of data-intensive multimedia applications involving large objects.

Acknowledgements. Results reported in the validation part of the cost model (see Sect. 6) are based on an actual implementation using NeoAccess by H.K. Wong and K.W. Yiu, who conducted this experimentation as part of their final year project on "Vertical Partitioning in Object Oriented Databases", Department of Computer Science, Hong Kong University of Science & Technology, Dec 1996.

References

1. Atkinson M, Bancilhon F, DeWitt D, Dittrich K, Maier D, Zdonik S (1989) The object-oriented database system manifesto. In: Proc. 1st International Conference on Deductive, Object-Oriented Databases, Kyoto, Japan, pp 40–57
2. Bertino E, Kim W (1989) Indexing technique for queries on nested objects. IEEE Trans Knowl Data Eng 1(2):196–214
3. Cluet S, Delobel C (1992) A general framework for the optimization of object-oriented queries. In: Proc. ACM-SIGMOD International Conference on Management of Data, pp 383–392
4. Chakravarthy S, Muthuraj J, Varadarajan R, Navathe SB (1994) An objective function for vertically partitioning relations in distributed databases and its analysis. Distrib Parallel Databases 2(2):183–207
5. Cornell DW, Yu PS (1987) A vertical partitioning algorithm for relational databases. In: Proc. 3rd International Conference on Data Engineering, Feb. 3-5, Los Angeles, Calif., USA, pp 30-35
6. Ezeife C, Barker K (1998) Distributed object-based design vertical fragmentation of classes. Parallel Distrib Database Syst 6(4):317–350
7. Fung CW, Karlapalem K, Li Q (1996) An analytical approach towards evaluating method-induced vertical partitioning algorithms. Technical Report HKUST-CS96-33, HKUST. Available at: http://www.cs.ust.hk
8. Fung CW, Karlapalem K, Li Q (1997) Cost-driven evaluation of vertical partitioning in object-oriented databases. In: Proc. 5th International Conference on Database Systems for Advanced Applications (DASFAA'97), Melbourne, Australia, pp 11–20, April 1–4. Extended version technical report HKUST-CS96-32, HKUST, available at: http://www.cs.ust.hk
9. Fung CW, Karlapalem K, Li Q (2002) An evaluation of vertical class partitioning for query processing in object-oriented databases. IEEE Trans Knowl Data Eng 14(5):1095–1118
10. Fung CW (1998) Vertical class partitioning and complex object retrieval in object-oriented databases. PhD Thesis, Department of Computer Science, Hong Kong University of Science & Technology
11. Gardarin G, Gruser JR, Tang ZH (1995) A cost model for clustered object-oriented databases. In: Proc. International Conference on Very Large Data Bases, pp 323–334
12. HERMES ESPRIT joint project group (1998) The HERMES Reference Architecture and Design Report. Technical report, available at: http://www.ced.tuc.gr/Research/Reports/HERMES/ Reports.htm
13. Hellerstein JM, Stonebraker M (1993) Predicate migration: optimizing queries with expensive predicates. SIGMOD Rec 2(2):267–277
14. Keller T, Graefe G, Maier D (19901) Efficient assembly of complex objects. In: Proc. of ACM-SIGMOD International Conference on Management of Data, pp 148–157
15. Kim W (1990) Introduction to object-oriented databases. MIT Press, 1990
16. Kim W, Lochovsky FH (1989) Object-oriented concepts, databases, and applications. ACM, New York
17. Karlapalem K, Li Q (1995) Partitioning schemes for object oriented databases. In: Proc. 5th International Workshop on Research Issues in Data Engineering – Distributed Object Management (RIDE-DOM'95), Taipei, Taiwan, pp 42–49
18. Karlapalem K, Li Q (2000) A framework for class partitioning in object-oriented databases. Distrib Parallel Databases 8:317–350
19. Karlapalem K, Li Q, Vieweg S (1996) Method-induced partitioning schemes in object-oriented databases. In: 16th International Conference on Distributed Computing System (ICDCS '96), Hong Kong, May, pp 27–30
20. Kemper A, Moerkotte G (1990) Access support in object bases. In: ACM SIGMOD International Conference on Management of Data, Atlantic City, N.J., USA, pp 364–374
21. Karlapalem K, Navathe SB, Morsi MA (1994) Issues in distribution design of object-oriented databases. In: M.T. Ozsu, U. Dayal, P. Valduriez (eds) Distributed object management. Morgan-Kauffman, San Francisco, pp 148–164
22. Navathe SB, Ceri S, Wiederhold G, Dou J (1984) Vertical partitioning algorithms for database design. ACM Trans Database Syst 9(4):680- 710
23. Neologic Systems (1996) NeoAccess Developer's Reference Manual. 1450 Fourth Street, Suite 12, Berkeley, CA 94710, USA
24. Navathe SB, Ra M (1989) Vertical partitioning for database design: a graphical algorithm. (1989) In: Proc. ACM SIGMOD International Conference on Management of Data
25. Ozsu MT, Blakeley JA (1995) Query processing in object-oriented database systems. In: W. Kim (ed.) Modern database systems: the object model, interoperability, and beyond, ACM, New York, pp 146–174
26. Ozsu MT, Valduriez P (1999) Principles of distributed database systems, 2nd edn. Prentice-Hall, Englewood Cliffs, N.J., USA
27. Straube DD, Ozsu MT (1995) Query optimization and execution plan generation in object-oriented data management systems. IEEE Trans Knowl Data Eng 7(2):210–227
28. Xie Z (1993) Optimization of object queries containing encapsulated methods. In: Proc. International Conference on Information and Knowledge Management, pp 451–460
29. Yao SB (1977) Approximating the number of accesses in database organizations. Comm ACM 20(4):260

Appendix A – [24] Graph-based algorithm

```
Step 1: Construct the affinity graph of
        the attributes in the classes
Step 2: Start from any node
Step 3: Select an edge which satisfies
        the following conditions: it
        should be linearly connected to
        the tree already constructed and
        it should have the largest value
        among the possible choices of
        edges at each end of the tree
        /* this iteration will end when
        all nodes are used for tree
        construction */
Step 4: When the next selected edge
        forms a primitive cycle:
        If a cycle node does not exist,
        check for the possibility of a
        cycle and if the possibility
        exists, mark the cycle as an
        affinity cycle. Consider this
        cycle as a candidate partition.
        Goto Step 3 If a cycle node
        exists already, discard this
        edge and goto Step 3
```

```
Step 5: When the next selected edge does
        not form a cycle and a candidate
        partition exists:
        If no former edge exists, check
        for the possibility of extension
        of the cycle by this new edge.
        If there is no possibility, cut
        this edge and consider the cycle
        as a partition. Goto Step 3 If a
        former edge exists, change the
        cycle node and check for the
        possibility of extension of the
        cycle by the former edge. If
        there is no possibility, cut the
        former edge and consider the
        cycle as a partition. Goto Step
        3
```

[24] graph-based algorithm