Cost-Driven Evaluation of Vertical Class Partitioning in Object Oriented Databases

Chi-wai Fung  Kamalakar Karlapalem  Qing Li
Department of Computer Science, University of Science and Technology
Clear Water Bay, Kowloon, Hong Kong, E-mail:(cwfung,kamal,qing)@cs.ust.hk

Abstract
Vertical partitioning is a proven database design technique which increases the efficiency of query execution by reducing the irrelevant data accesses. In this paper, we develop a cost model to study the effectiveness of vertical partitioning in OODBs, in terms of reducing the number of disk accesses for executing a set of queries. Further, we compare and evaluate two different (namely, affinity-based, and cost-driven) partitioning approaches. We show that the cost-driven approach is more general and more effective in reducing number of irrelevant disk accesses accessed than the affinity-based approach.

1 Introduction
Object oriented database (OODB) technology is being used to support main stream business information systems and decision support systems. In both these systems it is critical not only to have efficient implementation of database system but also a good database design. Any partitioning scheme for OODB system must take into consideration the characteristics of the OODB system. We motivated the need for class partitioning by enumerating and evaluating the issues involved in [9], developed, classified and represented different partitioning schemes in [10], and presented a method induced methodology for class partitioning and support for fragmentation transparency in [11]. This background work facilitated in-depth understanding of class partitioning for OODBs. In contrast to partitioning, indexing is a facility in OODBs to reduce the number of disk I/Os in query execution (e.g., see [1, 8]) by reducing the accesses to irrelevant object instances (as compared with sequential scanning). Indexing reduces disk I/Os at the object instance level; it still accesses irrelevant instance variables, as not all the instance variables accessed may be relevant to the query. Partitioning is also a facility in OODBs to reduce the number of disk I/Os in query execution by reducing the accesses to irrelevant fragments (that contains irrelevant instance variables). Partitioning reduces disk I/Os at the instance variable level.

The preliminary ideas about vertical class partitioning in OODBs were developed in [9, 10, 11]. In [10, 11], we developed representation schemes for vertical fragments, presented a method induced partitioning methodology for vertical partitioning, and presented a solution for supporting method transparency in OODBs. In [5], the authors presented an affinity-based approach [12] to vertical partitioning of object databases. However, they did not consider any representation scheme for vertical class fragments, and disregarded the best affinity based solution [3] for vertical partitioning. Further, they ignored the physical costs corresponding to the savings in the amount of irrelevant data accessed, and the overhead due to vertical partitioning. The utility of the vertical partitioning schemes [4] can be measured by the amount of savings in disk accesses for query execution. Thus, there is a need to develop a cost-driven approach based on the savings in disk accesses so as to evaluate the effectiveness of vertical partitioning in OODBs. While some partitioning algorithms have been proposed for OODBs, not much published work is on the quantitative query execution cost analysis. Although significant research has been conducted within the context of cost model for query execution in unpartitioned scenario (e.g., see [7]), there is not much detailed cost analysis on query execution for the vertically partitioned OODBMS. In order to fill this gap, this paper presents such a cost model and evaluates its utilities within an example OODB implementation. We further compare two contrasting approaches to vertical partitioning (namely, affinity-based and cost-driven).

The main contributions of this paper are: (1) Development of a cost model for executing queries on both vertically partitioned class collection and unpartitioned class collection; (2) Evaluation of utility of vertical partitioning based on the number of fragments, fan-out across class composition hierarchy, and cardinality of classes; (3) Development of two vertical partitioning techniques (cost-driven and affinity based) for OODBs and (4) Comparative evaluation of the above two vertical partitioning techniques.

The rest of the paper is organized as follows: Section 2 presents a cost model for vertical partitioning. Section 3 evaluates the cost model and presents results of experiments conducted to show the benefit of vertical partitioning. Section 4 presents two techniques for vertical partitioning in OODBs. Section 5 presents a detailed example illustrating both the techniques, and Section 6 presents the conclusions.

2 Cost model for Vertical Partitioning (VP)
In this section, we present a general analytical cost model for executing a query for both the unpartitioned class and the vertical partitioned class.

The total cost to execute a query is given by:

\[ Total\_cost = IO\_cost + CPU\_cost \]

where \( IO\_cost \) is the cost for performing disk I/O and \( CPU\_cost \) is the cost for performing the computation during the query execution. In this paper, as in [4], we concentrate on the \( IO\_cost \) 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 \( Total\_cost \) will not be significant.

We assume a syntax of Object Query Language (an Object Oriented version of SQL) as given below.
SELECT "result list"
    FROM "target class"
WHERE "condition/predicate"

The "result list" involves only instance variables from a single class. Also, other than the path expression, the "condition/predicate" clause, can only involve instance variables from the root class of the class hierarchy. The path expression is of the form
\[ C_1 \cdot a_1 \cdot a_2 \cdot \ldots \cdot a_n \cdot \text{vbi} \text{v relop const} \],
where \( a_i \in C_{i-1} \) with \( 2 \leq i \leq n \), i.e., \( a_i \) is the object based instance variable defined in class \( C_{i-1} \) with domain of class \( C_i \) evaluated as an implicit join condition is of the form \( C_1 \cdot a_1 \cdot a_2 \cdot \ldots \cdot a_{n-1} \cdot \text{vbi} \text{v relop const} \), where \( \text{relop} \) can be any relational operator: \( =, \prec, >, \ll, \lll \) and \( \text{const} \) is a constant within the domain of the value based instance variable \( \text{vbi} \text{v} \) of class \( C_a \). The cost model is based on a set of parameters which can be categorized into three types shown in Table 1, namely, database system parameters (such as, cardinality of a class), transactional parameters (such as, selectivity of the query), and specific vertical partitioning parameters (such as, number of fragments).

We make the following observations regarding the amount of main memory, as main memory effects the number of disk accesses: (1) 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) and (2) 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 the predicate evaluation, the same objects or object fragments of a particular class may be required to be loaded into the main memory multiple times and cause a high increase in the number of disk I/O).

2.1 Cost of executing a query on unpartitioned classes

Consider the class hierarchy for an unpartitioned classes case. There is an IsPartOf/composition class hierarchy along the path from class \( C_1, C_2, \ldots, C_n \) (and all these classes are unpartitioned). For every class \( C_j \), there is also an ISA class hierarchy rooted by it. Our cost model is quite general in that these different ISA hierarchies may have different fan-outs from parent node to child nodes, they can also have different maximum tree levels in the hierarchy. We assume that all the subclasses of a class are stored together with the parent root class (this assumption can be relaxed, please refer to [7] for an alternate formulation). The cost model can be broken up into 3 components: the cost of loading a class collection, the cost of evaluating the predicate, and the cost of building the output result.

The unpartitioned class hierarchy cost model is similar to [7]'s formulation.

(a) Estimate of the number of pages in a class collection

Sequential scan and index scan are the two major strategies used when scanning a class collection. The objective of using index is to attain faster object instance access, while the objective of using vertical partitioning is to reduce irrelevant instance variable accesses. Hence the two objectives are orthogonal and complementary. In this paper, we shall concentrate on the sequential scan strategy; the use of index can also be incorporated into our model naturally as needed. The objects of a class are assumed to be stored/accessed on/from the disk sequentially. Also, the objects are assumed not to cross page boundaries. The total number of pages occupied by a class collection \( C \) with object length/size \( L_C \) is given by: \[ P_C = \left\lceil \frac{\text{card} (C)}{L_C} \right\rceil \]

where \( \lceil \rceil \) and \( \lfloor \rfloor \) are the ceiling and floor functions.

(b) Cost formulae for query execution

For the LMH case, the \( \text{IO Loading Cost} \) (first term) is the cardinality of the root class ISA hierarchy divided by the number of objects per page and the \( \text{IO Eval Cost} \) (second term) will be the sum of Yao functions as in [13]. Further, the \( \text{IO Building Output} \) (third term) is the cardinality of the output result collection (i.e., the selectivity times the cardinality of the root class ISA hierarchy) divided by the number of output objects per page. So the total IO cost of the unpartitioned case is given by:

\[
\text{Total IO UP LMH} = \text{IO Loading Cost} + \text{IO Eval Cost} + \text{IO Building Output}
\]

For the SMH case, the \( \text{IO Loading Cost} \) and \( \text{IO Building Output} \) are identical to that of the LMH case. The \( \text{IO Eval Cost} \) is quite different from LMH case:

\[
\text{IO Eval Cost} = \sum_{i=2}^{n} \text{card}(C_i) \cdot \text{card}(C_i) \cdot \text{ref}_i \cdot \text{ps} \cdot \text{proj}
\]

as predicates involve only instance variables of class \( C_1 \), we have:

\[ \text{ref}_1 = [C_i]_{\text{Hier}}, \text{ref}_2 = \text{ref}_1 \times \text{sel} \times \text{fan}_{1,2} \text{ and ref}_i = \text{ref}_{i-1} \times \text{fan}_{i-1,i}, \text{ (sel} = 1.0, \text{ if not specified) for i > 2.}\]

For the SMH case, the \( \text{IO Loading Cost} \) and \( \text{IO Building Output} \) are identical to that of the LMH case. The \( \text{IO Eval Cost} \) is quite different from LMH case:

\[
\text{IO Eval Cost} = \sum_{i=2}^{n} \text{card}(C_i) \cdot \text{card}(C_i) \cdot \text{ref}_i \cdot \text{ps} \cdot \text{proj}
\]

the cost of evaluating the predicate is a sum of page accesses for the different class collections along the path expression. In the ith class collection along the path expression, we need \( \text{card}(C_i) \cdot \text{card}(C_i) \cdot \text{ref}_i \cdot \text{ps} \cdot \text{proj} \) page accesses. Thus, SMH requires many more pages to be loaded into main memory than LMH. Therefore, the total IO cost for SMH is given by:

\[ Y_{\text{eval}}(n,b,k) = b \cdot \frac{n}{2} \cdot \prod_{i=1}^{n} \frac{n-i+1}{n-i+1} \]

where \( d=1/(4b) \). The expected number of page accesses is not equal to \( k \) because some pages may contain 2 or more goal records. In our discussion, we usually take \( n = |C|, b = |C| \) and \( k = \text{sel} \times |C| \).
2.2 Cost of executing a query on vertically partitioned classes

We adopt an object oriented representation of a vertical partitioning of a class [10]: each of the vertical class fragments is represented as a class, and a logical object of class C is internally represented as a composite object (of class C") that consists of objects from the vertical class fragments.

![Figure 1: A Vertically Class Partitioned Object Oriented Database Schema](image)

Figure 1 shows a vertically partitioned class hierarchy. There is an IsPartOf/composition class hierarchy along the path from class C_1, C_j, through C_n. For every class C_j, there is also an ISA class hierarchy rooted by it. Some of the classes/subclasses are further vertically partitioned. Concerning the vertical partitioning, not all classes along the path need to be vertically partitioned. We define binary variables VP_i, where

\[ VP_i = \begin{cases} 
1 & \text{if class } C_i \text{ is vertically partitioned} \\
0 & \text{otherwise}
\end{cases} \]

where 1 ≤ i ≤ n. As elaborated in [10] if a class C_i (along a path) is vertically partitioned into m_j vertical class fragments C_iV_j through C_iVm_j, then the subclass of class C_i say C_k, is also vertically partitioned into m_j vertical class fragments C_kV_j through C_kVm_j.

Given a predicate/condition in a query, we define binary variables P_j, where

\[ P_j = \begin{cases} 
1 & \text{if fragment class } C_jV_j, \text{ contains some instance variables class are involved in the predicate} \\
0 & \text{otherwise}
\end{cases} \]

where 1 ≤ i ≤ n. We further define, binary variables RL_j, where

\[ RL_j = \begin{cases} 
1 & \text{if fragmented class } C_jV_j, \text{ contains some instance variables in the result list} \\
0 & \text{otherwise}
\end{cases} \]

where 1 ≤ i ≤ n. For the LMH case, for loading the class hierarchy, we need only to load in pages that are related to either the predicate or the result list. As there are m_j fragments, the IO_Loading_Cost is a summation over the number of pages occupied by the fragments with \( P_j = 1 \) or \( RL_j = 1 \), where 1 ≤ j ≤ m_j. Just like the unpartitioned case, the IO_Eval_Cost is a sum of Yao functions over the composition class hierarchy, but now we need to consider the effect of vertical partitioning of the classes along the path (which is indicated by the binary variables VP_i). Finally, the IO_Building_Output is also a summation over the fragments that give the result fragment pages. Therefore, the total IO cost of the vertical partitioned case is given by:

\[ \text{Total}_{-IO}\_UP\_SMH} \]

\[ \begin{align*} 
&= \left[ I\_C\_Hier \cdot PS \right] + \sum_{i=1}^{n} \left[ I\_C\_Hier \cdot PS \cdot \text{Fan}_{i} \right] \\
&+ \left[ \text{SEL} \cdot I\_C\_Hier \cdot PS \cdot \text{Lproj} \right]
\end{align*} \]

<table>
<thead>
<tr>
<th>Category</th>
<th>Parameter</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Database</td>
<td>C</td>
<td>cardinality of a class collection C (i.e., a number of objects)</td>
</tr>
<tr>
<td></td>
<td>C_n</td>
<td>total number of pages occupied by class C</td>
</tr>
<tr>
<td></td>
<td>LC</td>
<td>object length/size (number of bytes) in class collection C</td>
</tr>
<tr>
<td></td>
<td>C_iHier</td>
<td>cardinality of the whole ISA hierarchy rooted at class C_i</td>
</tr>
<tr>
<td></td>
<td>C_iHier</td>
<td>total number of pages occupied by the class ISA hierarchy rooted at class C_i</td>
</tr>
<tr>
<td></td>
<td>LC_i</td>
<td>object length/size in class collection C_i</td>
</tr>
<tr>
<td></td>
<td>Fan_{i}</td>
<td>the fan-out for the class composition hierarchy from class C_{i+1} to C_i</td>
</tr>
<tr>
<td></td>
<td>n</td>
<td>the path length of the class composition hierarchy, i.e., the number of classes along the path</td>
</tr>
<tr>
<td></td>
<td>PS</td>
<td>the page size of the file system (in unit of byte)</td>
</tr>
<tr>
<td>Query</td>
<td>RSP_{i}</td>
<td>the number of object references for class C_i during the path expression evaluation process along the class composition hierarchy</td>
</tr>
<tr>
<td></td>
<td>SEL</td>
<td>selectivity of the predicate</td>
</tr>
<tr>
<td></td>
<td>Lproj</td>
<td>the length of output result that is within class C_1</td>
</tr>
<tr>
<td></td>
<td>P_j</td>
<td>a binary variable, it is of value 1 if the predicate accesses fragment j in the root class C_1, 0 otherwise</td>
</tr>
<tr>
<td></td>
<td>RL_j</td>
<td>a binary variable, it is of value 1 if the result list accesses fragment j in the root class C_1, 0 otherwise</td>
</tr>
<tr>
<td></td>
<td>Lproj_j</td>
<td>the length of output result that is within fragmented class C_j ( V_j ), 0 otherwise</td>
</tr>
<tr>
<td>Vertical</td>
<td>m_i</td>
<td>the number of fragments in class C_i</td>
</tr>
<tr>
<td>Partitioning</td>
<td>C_iV_iHier</td>
<td>cardinality of the whole class collection ISA hierarchy rooted at vertical partitioned class C_i ( V_i )</td>
</tr>
<tr>
<td></td>
<td>C_iV_iHier</td>
<td>total number of pages occupied by the whole class collection ISA hierarchy rooted at vertical partitioned class C_i ( V_i )</td>
</tr>
<tr>
<td></td>
<td>LC_iV_i</td>
<td>object length/size in the fragmented class collection C_i ( V_i )</td>
</tr>
<tr>
<td></td>
<td>VP_i</td>
<td>a binary variable, it is of value 1 if class C_i along the class composition hierarchy is vertically partitioned, 0 otherwise</td>
</tr>
</tbody>
</table>

2. Without loss of generality, for a vertically partitioned class C_j, the first fragment contains the object-based instance variable forming the class composition hierarchy.
For the SMH case, the IO-Loading-Cost and IO-Building-Output are identical with that of the LMH case. The IO-EvaI-Cost is quite different from LMH case, but is identical to the unpartitioned SMH case:

\[
\text{IO-EvaI-Cost} = \|C,Hier\| \cdot \left( \text{IFAN}_{i,j} \cdot \text{REF} \right)
\]

If we do not have enough memory, the cost of evaluating the predicate is the same as the unpartitioned SMH case, that is, the vertical partitioning does not have any performance gain compared to the unpartitioned case for evaluating the predicate. So the total IO cost of the vertical partitioned case is given by:

\[
\text{Total IO-VP-SMH} = \sum_{i} (\|P_{i} \times RL_{i}\| \cdot \text{LC}_{i,j}) + \sum_{i} \left( \|P_{i} \times RL_{i}\| \cdot \text{PS} \cdot \text{SEF}_{i} \cdot \|C,Hier\| \right)
\]

3 Evaluation of vertical partitioning schemes

In this section, we compare the cost formulae for unpartitioned case with the vertical partitioned case to evaluate the utility of vertical partitioning. Then, we suggest some guidelines for the use of vertical partitioning. We shall compare these cost formulae component by component.

3.1 Comparison of total IO cost for unpartitioned and partitioned classes

We concentrate in the discussion of the LMH case first.

(a) LMH case

Loading of class hierarchy: after taking some approximation (by ignoring the effects of the ceiling and floor functions), vertical partitioning will benefit the query processing if \( \sum_{i} (\|P_{i} \times RL_{i}\| \cdot \text{LC}_{i,j}) \) is less than \( \text{LC}_{c} \). That means a good vertical partitioning scheme should minimize \( \sum_{i} (\|P_{i} \times RL_{i}\| \cdot \text{LC}_{i,j}) \). This can be done by either minimizing \( \text{LC}_{i,j} \), i.e., the length of the vertical fragments involved in the predicate or result list should be as short as possible. Or, we can minimize \( \sum_{i} (\|P_{i} \times RL_{i}\| \cdot \text{LC}_{i,j}) \), i.e., we should try to group the instance variables involved in the predicate and result list into as few vertical fragments as possible. But note that vertical partitioning may not always improve the performance. In particular, if the classes are highly fragmented into many very small fragments, one cannot ignore the effect of the ceiling function, because each small fragment (no matter how small) still requires one disk IO access. In conclusion, if a class is partitioned into a large number of small fragments, it will have negative effect on the IO-Loading Cost.

Predicate evaluation: as the length of a vertical fragment must be smaller than the original unpartitioned object, i.e., \( \text{LC}_{i,j} \) is always less than \( \|C,Hier\| \) for vertically partitioned classes, and

\[
\text{Total IO-VP-LMH} = \sum_{i} (\|P_{i} \times RL_{i}\| \cdot \|C,Hier\|) + \sum_{i} \left( \|P_{i} \times RL_{i}\| \cdot \text{PS} \cdot \text{SEF}_{i} \cdot \|C,Hier\| \right)
\]

That is, vertical partitioning will always improve the performance of evaluating the predicate. If the fan-out between the classes in the class composition hierarchy is high, the IO-Eval-Cost component will dominate the total IO cost. Furthermore, it is advisable to use vertical partitioning along the whole path expression, especially for class composition hierarchy with high fan-out.

Building the output result: as \( \text{IO-Cost} = \sum_{i} (\|P_{i} \times RL_{i}\| \cdot \text{PS}) \), the IO cost of building the output in both cases can be treated as the same (an approximation by neglecting the effects of the ceiling and floor functions). So vertical partitioning will not improve the query execution during the output building phase.

(b) SMH case

For the SMH case, the vertical partitioned and the unpartitioned cost formulae only differ in the IO-Loading-Cost component (the IO-Eval Costs are identical in both cases and the IO-Building Outputs are approximately equal). For IO-Loading Cost, the comparisons are similar to those given in Loading of class hierarchy in LMH case.

3.2 Experiments to evaluate the utility of vertical partitioning

(a) Performance metric

To better compare and contrast the utility of vertical partitioning, we conducted a few experiments to evaluate the effect of number of vertical fragments, the fan-out, and the cardinality on the improvement of performance due to vertical partitioning in LMH case. This is because SMH is a restricted case of LMH, and taking into consideration the current computer technology, wherein most systems have large main memory, LMH is more realistic than SMH. This improvement of performance is characterized by the normalized IO metric.

\[
\text{Normalized IO} = \frac{\# \text{ of IOs for vertically partitioned class collection}}{\# \text{ of IOs for unpartitioned class collection}}
\]

If the value of Normalized IO is less than 1.0, then it implies that vertical partitioning is beneficial. In order to present interesting results, while maintaining the control over the number of parameters and studying the impact of parameter changes, we consider the following seven parameters. These are cardinality of root class, page size, number of objects per page, number of vertical fragments per class, fan-out of a class along class composition hierarchy, selectivity of predicate, and projection ratio of a query. Further, in these experiments, we consider a class collection with a class composition hierarchy of path length as 3. That is, there are three classes \( C_{1}, C_{2}, \) and \( C_{3}, \) with a class composition hierarchy \( C_{1} \rightarrow C_{2} \rightarrow C_{3} \). We study the
following cases of vertical partitioning:
- **VP1** - only class $C_1$ is vertically partitioned;
- **VP2** - both classes $C_1$ and $C_2$ are vertically partitioned;
- **VP3** - all the classes $C_1, C_2$ and $C_3$ are vertically partitioned.

The reason for selecting such a class collection being that it enables us to study both the impact of fanout along the class composition hierarchy, and also the effect of partitioning classes along class composition hierarchy. Note that a query accesses the objects of the target class based on the predicates which can restrict the objects along the class-composition hierarchy paths rooted at the target class. Therefore, we need to evaluate only the vertical partitioning of classes along the class composition hierarchy. Otherwise, ad-hoc partitioning of classes without considering the queries that access objects based on path expressions will be meaningless.

(b) **Experiment parameters**

The parameter settings are as follows: the cardinalities of the three classes are given by $[C_{1}]$, $[C_{2}] = [C_{1}] \times FAN_{1,2}$ and $[C_{3}] = [C_{2}] \times FAN_{2,3}$, where $FAN_{1,2} = FAN_{2,3}$, the length of objects for all the three classes is the same, the length of fragments is the same (i.e., we divide each class into equal length fragments). Furthermore, unless specified otherwise the classes are fragmented into same number of vertical class fragments. The projection ratio defined as the ratio between the length of the requested output result (some projection on the original class $C_1$ objects) and the length of class $C_1$ object. In the cost model, we need to obtain the values for the logical "OR" sum (denoted as ORSUM) of $P_j \land RL_j$ over all fragments. Without any query characteristics/information, we have the following simplification in calculating ORSUM of $P_j \land RL_j$: if $m$ is the number of fragments, the ORSUM should be a value between 1 and $m$. ORSUM of value 1 is not always possible. This is because for large projection ratio ($PR$), one fragment cannot contain all the result instance variables. We use the formula $ORSUM = ceiling( m \times PR )$. If $m \times PR$ is small (say 0.4) we use the ceiling function to make up the ORSUM to be at least 1. On the other hand, if $PR$ is large (say 0.99) we have the other extreme that the ORSUM is just $m$. The motivation of setting these parameter values is to present the causes for interesting results, while not being tied down by too many parameter values.

(c) **Effect of varying the number of fragments**

In this experiment we study the impact of number of fragments on the performance gain due to vertical partitioning. The result of this experiment is to show the existence of optimal number of vertical class fragments for a class. The parameter values used to produce experimental results shown in Figure 2 are: cardinality of $C_1$, class hierarchy is 1000, page size of 4096 bytes, the fan-out is 1.0, the projection ratio is 0.8 and the number of objects per page is 16. These parameters are similar to those used by the O07 benchmark [2]. In Figure 2, all curves show a minimum normalized IO when the optimal number of fragments equals to 64. The performance gain for different number of fragments ranges from the best 63% (0.63=1-0.37, for VP3 with medium (0.5) selectivity) to the worst -104% (-1.04=1-2.04, for VP1 with low (0.05) selectivity). The above results imply that the optimal number of fragments exists and a good choice of the number of fragments can produce an optimal partition scheme with high performance gain. The plots for other numbers of objects per page show similar results. Note that the plots of Normalized IO vs. Number of Fragments show that at first the normalized IO decreases for increase in the number of fragments, but starts to rise as the number of fragments further increased, giving an optimal number of vertical fragments. This can be explained by the fact that the vertical fragments are stored independently, i.e., different class fragments are stored in different disk pages. Large number of vertical fragments leads to small fragments, and accessing large number of small fragments causes extra disk IO. Therefore, the performance gain will decrease (due to overhead) and the normalized IO value will start to increase. In some cases, the normalized IO value for very large number of fragments can shoot up above 2 as shown in Figure 2 resulting in a poorer performance than the unpartitioned case.

(d) **Effect of varying the fan-out**

In this experiment we study the impact of the variations in the fan-out (along class composition hierarchy) in a class hierarchy on the performance gain. The main result of this experiment is to show the performance gain for high fan-out values. Parameter values used to produce experimental results shown in Figure 3 are: cardinality of $C_1$, hierarchy is 1000, page size 4096 bytes, the number of objects per page is 16, the projection ratio is 0.8 and the number of fragments in class $C_1$ is 32. We observe following trends: the lower bound of normalized IO approaches 1/(number of fragments), for VP3 curves at high fan-out. That is, great savings can be obtained by using vertical partitioning at high fan-out. In Figure 3, the number of fragments in class $C_1$ is 32 and in the curves of VP3, the normalized IOs approach $1/32 = 0.03$, that is, a performance gain of 1-0.03 = 97% saving, which is quite substantial in query execution. Note that with high fan out, the number of object instances in classes $C_2$ and $C_3$, i.e., $[C_{2}]$ and $[C_{3}]$, will be quite large when compared to $[C_{1}]$. Therefore, the $IO\_Eval\_Cost$ will dominate the total IO cost. For the VP3 case, all classes in $C_1$, $C_2$ and $C_3$ are vertically partitioned into 32 fragments, that is, during the predicate evaluation, we need only to retrieve 1 out of the 32 fragments, from classes $C_2$ and $C_3$ (we only need the fragment that contains the object based instance variable along the path).

(e) **Effect of varying the cardinality**

In this experiment, we study the impact of the variations of the cardinality of the root class on the performance gain. The main result of this experiment is to show the performance gain in terms of normalized IO is constant as the cardinality increases. Parameter values used to produce experimental results shown in Figure 4 are: cardinality of $C_1$ class hierarchy is varied from 1000 to 1000000, page size 4096 bytes, the number of objects per page is 16, the projection ratio is 0.2, the fan-out is 1.0 and the number of fragments in $C_1$ is
16. All the curves in Figure 4 are almost horizontal straight lines, showing that the normalized IO is independent of cardinality of the root class. That is, irrespective of the cardinality of the class we get the same percentage of reduction in number of disk accesses.

(f) Summary of results from experiments

The experiments that we conducted present us with following results: (1) There is an optimal number of vertical fragments for a class collection, and vertical partitioning can give rise to substantial savings in number of disk accesses; (2) The fan-out parameter has an impact on the vertical partition algorithms. If there are queries that access objects using path expressions, then it is preferable to vertically fragment all the classes in the class composition hierarchy. This results in considerable IO savings for low fan-outs with the IO savings increasing with the increase in the fan-out and (3) The cardinality of the class does not impact the normalized IO savings, implying that as the cardinality of the classes increases the IO savings will also proportionally increase.

Based on the above results we foresee a need for cost-driven algorithm for vertical partitioning of class collection. Further, vertical partitioning is beneficial when the number of fragments is small. In the next section, we present two contrasting approaches to vertical partitioning.

4 Vertical partitioning techniques

There are two main techniques for vertical partitioning, namely, affinity-based [3, 12] and cost-driven [4]. In [3], it was shown that the optimal partition scheme generated by different affinity-based partitioning techniques are sometimes different even for the same input. Furthermore, an "objective" affinity-based cost model was developed to evaluate the performance of different affinity-based partitioning techniques. It was shown that their Minimum Square Error (MSE) performed the best. Though this approach was developed for relational database systems, it is equally applicable for object oriented databases. This MSE approach will be compared against the cost-driven approach developed in this paper. Note that we are evaluating two techniques for vertical partitioning, and we do this by exhaustively enumerating all the possible vertical fragments and finding the best vertical partition generated by both the approaches. The MSE partitioning technique can at best equal the number of disk accesses given by the optimal partitioning scheme generated by the cost-driven technique.

4.1 Minimum Square Error (MSE) technique

The input is an attribute usage matrix which consists of the attribute in a relation as columns and the queries as rows with the frequency of access to the attribute for each query as the values in the matrix. In this model, there are two terms to measure the goodness of the vertical partition scheme, one is the irrelevant attribute access cost and the other one is the relevant attribute access cost. The irrelevant attribute access cost is represented by the square error between the actual access patterns of the query in the partition scheme and the mean vector (which represents the average access pattern of the queries over all attributes of the fragments). The relevant access cost computes a penalty factor that measures the contribution due to access of relevant attributes that are dispersed in different fragments. We shall use the following notations:

- \( A_{ij} \) is the attribute vector for attribute \( j \) in fragment \( i \).
- \( V_i \) is the mean vector for the \( i \) th fragment. It represents an average access pattern of the queries over all attributes of fragment \( i \).
- \( n_i \) is the number of attributes in fragment \( i \).
- \( M \) is the total number of fragments of a class.
- \( T \) is the vector transpose operation.
- \( NTX \) is the total number of queries that are under consideration.
- \( freq \) is the frequency of query \( t \).
- \( |R_{ikt}| \) is the number of relevant attributes in fragment \( k \) accessed with respect to fragment \( i \) by query \( t \).
- \( n_{ikt} \) is the total number of attributes that are in fragment \( k \) accessed with respect to fragment \( i \) by query \( t \).
- \( E^1 \) is the irrelevant attribute access cost.
- \( E^2 \) is the relevant attribute access cost.
- \( E^3 \) is the square error.

The cost formula for irrelevant attribute access cost:

\[
E^1 = \sum_{i=1}^{M} \sum_{j=1}^{n_i} (A_{ij} - V_i)^T (A_{ij} - V_i)
\]

where \( V_i = \frac{1}{N} \sum_{j=1}^{n_i} A_{ij} \)

The cost formula for relevant attribute access cost:

\[
E^2 = \sum_{i=1}^{M} \min_{k=1}^{N_{ITX}} \left( \sum_{t=1}^{freq} (|R_{ikt}| \cdot n_{ikt}) \right)
\]

The overall Square Error is given by:

\[
E^3 = E^1 + E^2
\]

An MSE vertical partitioning procedure

With an input of attribute usage matrix, the following algorithm will be used to enumerate all possible vertical partitioning schemes to find the optimal vertical partition schemes for all classes in the schema:

Step 1. For each class in the schema Do
Step 2. For \( 1 \leq i \leq \text{NumberOfInstanceVariable} \) in that class Do
Step 3. Enumerate all \( i \) fragment(s) partition schemes
Step 4. Calculate the Square Error for each partition scheme and determine the min Square Error partition scheme
Step 5. EndFor
Step 6. From all of the above min square Error i fragment partition schemes, determine the final overall min Square Error partition scheme
Step 7. EndFor

4.2 Cost-Driven Vertical Partitioning (CVP) technique

In this approach, we present a cost model based on the number of disk accesses taken to execute a query. We evaluate the partitioning schemes based on this performance metric. In our model, we are not only considering query parameters (like access frequency), but also consider database characteristics (like the length of the instance variables and fan-out) and other parameters like selectivity of query, and the instance variables participating in the query predicates and/or the query output result list. The general cost formula with query parameters is:
that is, the total IO cost is a weighted sum of IO cost over all queries with weighting factor \(freq_i\). For the cost model components please refer to [6] for details.

A CVP vertical partitioning procedure

The basic strategy is to start with the leaf classes and vertical partition them first, then propagate the saving in irrelevant data accesses towards the root class of a class composition hierarchy.

Step 1. For each class (starting from the leaf classes to the root class of a class composition hierarchy) Do

Step 2. For \(i=1\) to \(NumberOfInstanceVariable\) in that class Do

Step 3. Enumerate all \(i\) fragment(s) partition schemes

Step 4. Use our cost model to calculate the total IO Cost for each partition scheme and determine the min cost partition scheme

Step 5. EndFor

Step 6. From all of the above min cost \(i\) fragment partition schemes determine the final overall min cost partition scheme

Step 7. EndFor

4.3 Comparison of our CVP technique with MSE technique

Both these techniques generate non-overlapping and complete vertical partitioning schemes. Unlike CVP technique the MSE technique does not take into account class hierarchy and class composition hierarchy. The MSE technique uses the instance variable usage frequency as the sole input, but CVP technique also takes care of the database characteristics (like instance variable length, fan-out and cardinality) and other query characteristics (like path length). The MSE technique assumes that any data access is of unit cost, but our CVP technique considers detailed cost breakdown at different stages in the query execution (such as, loading of class hierarchy, predicate evaluation, and result generation). Finally to obtain the optimal partitioning scheme, both techniques use the exhaustive search strategy to enumerate all the partitioning scheme and calculate their cost to obtain the minimum.

The CVP approach exploits the class composition hierarchy, in that, the non-leaf classes of class composition hierarchy can use the partitioning of their child classes to derive better vertical class fragments. But in MSE technique all classes are partitioned independently.

5 Example

We shall now take an example of a class collection with three classes (Emp, Dept, and Proj) and the class composition hierarchy of path length 3. The motivation of this experiment is to show the differences between MSE and CVP procedures in finding optimal partition scheme for the same example database.

```sql
Class Emp {
    DeptInfo
    EmpId int;
    DName Char[50];
    Skill Char[200];
    DAddress Char[100];
}
Class Dept {
   ProjInfo
    DeptId int;
    DName Char[50];
    DeptType Char[10];
    DAddress Char[100];
}
Class Proj {
    ProjInfo
    Priority Char[100];
    Location Char[100];
    PName Char[8];
    PId int;
    PName Char[50];
    PName Char[100];
}
```

5.1 Query characteristics

There are 18 queries, which are classified into 6 types, which will be used to compare both the MSE and CVP techniques for vertical partitioning. We show only the "Type 3" queries which consisted of 3 queries accessing all 3 classes and are "rooted" at the Emp class:

5.2 Minimum Square Error (MSE) technique

Attribute usage matrices on classes Emp, Dept and Proj are constructed from the above query characteristics. It is the only input required by the MSE vertical partitioning procedure. The results are: for class Proj, it is a 2 fragments partition scheme: (Priority Location) (ProjType PId PName); for class Dept, it is a 2 fragments partition scheme (ProjInfo DAddress) (DeptId DName DeptType); and for class Emp, it is a 2 fragments partition scheme (DeptInfo EAddress) (EmpId EName Skill). Figure 5(a) shows the optimal partition schemes under MSE for class Emp (the results for other classes, please refer to [6]). Figure 5(b) shows the scatter plot of Min Square Error for the experiment on class Emp for different partition schemes.

5.3 Cost-Driven Vertical Partitioning (CVP) technique

The experiment uses following parameter values: page size 4096 bytes, cardinality of root class Emp is 1000, length of an object based instance variable (logical OID) = 8 bytes and the fan-out from Emp to Dept class is 0.2 while fan-out from Dept to Proj class is 5.0. The other derived parameter values are calculated from these parameter values and the query characteristics listed above. When we execute our CVP vertical partitioning procedure, we have the following execution steps: the first step is to find the optimal leaf class Proj vertical partition scheme, while the second and third steps are to find the optimal class Dept and root class Emp partition schemes. The leaf class Proj is chosen, we enumerate all possible partition schemes to find the optimal partition scheme. In calculating the total IO cost, only transactions that access the Proj class are used. The optimal partition scheme is the 3 fragments partition scheme (PName) (ProjType PId) (Priority Location). The instance variable PName is involved in the path expression DeptInfo.ProjInfo.PName, the information that PName is in a fragment by itself (of length 50 bytes) is used in steps 2 and 3. In step 2, the class Dept is chosen, similar to the above, when calculating the total IO cost in this part, only transactions that access the Dept class (as "root"
class) are used. The optimal partition scheme is (DAddress) (ProjInfo DeptId) (DName DeptType). The instance variable ProjInfo is involved in the path expression DeptInfo.ProjInfo.PName, the information that ProjInfo is in a fragment with another instance variable - DeptId, with total length of 12 bytes long, is used in step 3. Finally in step 3, the root class Emp is chosen, similar to the above, when calculating the total IO cost in this part, only transactions that access the Emp class are used. The results are plotted in Figure 6(a). The optimal partition scheme is (DeptInfo) (Skill) (EAddress) (EmpId EName). As shown in Figure 6(a) the optimal partition scheme is a 4 fragments partition scheme. It is interesting to note that the values for cases with 3, 4 and 5 fragments are quite close, that means if the overhead of accessing multiple fragments is high, the partition scheme with 3 fragments can also be utilized (although it is not the optimal partition scheme) to improve performance. Figure 6(b) shows the scatter plot of total IO cost for different partition schemes. There is a wide spread in the values for the different partition schemes with the same number of fragments. For example, for 4 fragments partition schemes, the total IO costs range from 15276 to 22268 disk accesses. From this, we can conclude that we should be careful in choosing the vertical partition scheme. Just limiting number of fragments in a vertical partitioning scheme will not materialize in a performance gain.

5.4 Comparison of MSE technique vs. CVP technique

It is interesting to compare the values for these optimal partition schemes obtained from the two techniques. As shown Table 2, the values in brackets are not the optimal values in those vertical partitioning procedures, rather they are shown here to contrast the two different techniques. From Table 2, we can draw the conclusion that the two techniques produce quite different optimal partition schemes for database schema with same set of user queries and their access patterns. We do another set of experiments on class Emp, in this new set of experiments, we reduce the Skill instance variable length from 200 bytes to 1 byte and keep the other parameter values unchanged. As the MSE technique does not take into the account of instance variable length, so its optimal partitioning scheme remains unchanged. But the CVP technique takes into account of the instance variable length and produces a new optimal partition scheme: (DeptInfo) (EAddress) (EmpId EName Skill). Thus, CVP technique adapts to the changes in the physical database system parameters and generates the optimal partitioning scheme, whereas, MSE technique does not. Further, as shown in columns 4 and 6 of above table, the CVP technique presented the optimal partitioning solution which requires least number of disk accesses. At best, the MSE technique can generate a partition which requires minimum number of disk accesses, but this not always guaranteed as shown in this example. On the other hand, CVP technique always generates a partition scheme that minimizes the number of disk accesses.

6 Conclusions

Vertical partitioning in object oriented databases is a very challenging and relevant problem. Though this problem has been addressed in relational database systems, the complexity of object oriented data model with class hierarchy and class composition hierarchy present a need for new approach to this problem. There are fundamentally two different approaches to vertical partitioning, namely, affinity-based, and cost-based. In this paper, we developed the cost model for vertical partitioning, with the conclusion that optimal vertical partitioning does mostly reduce the cost of executing the query. Further, we developed two procedures for vertical partitioning based on affinity-based approach and cost-driven approach. We showed that the cost-driven approach is superior to the affinity-based approach in that it guarantees the minimum number of disk accesses to process all the queries.

References


Table 2. Table to compare the MSE and CVP experiment results

<table>
<thead>
<tr>
<th>Class</th>
<th>MSE optimal partition scheme</th>
<th>Corresponding IO cost</th>
<th>CVP optimal partition scheme</th>
<th>Corresponding IO cost</th>
<th>MSE optimal partition scheme</th>
<th>Corresponding IO cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>Emp</td>
<td>(DeptInfo EAddress) (EmpId EName Skill)</td>
<td>6102 (22441)</td>
<td>(DeptInfo) (Skill) (EAddress) (EmpId EName)</td>
<td>15276 (17177)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dept</td>
<td>(ProjInfo DAddress) (DeptId DName DeptType)</td>
<td>1331 (1415)</td>
<td>(DAddress) (ProjInfo DeptId) (DName DeptType)</td>
<td>1181 (4864)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Proj</td>
<td>(Priority Location) (ProjType PId PName)</td>
<td>625 (2775)</td>
<td>(PName) (ProjType PId) (Priority Location)</td>
<td>2725 (1875)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>


Plot of Min Square Error vs. Number of Fragments

Figure 5(a): Plot of Min Square Error vs. Number of Fragments to find the Optimal Emp class partition scheme

Calculation Part 3 to find the Optimal Emp class partition scheme

Scatter Plot of Square Error vs. Number of Fragment

Figure 5(b): Scatter Plot of Square Error vs. Number of Fragments for Emp class partition scheme

Plot of Min Total IO Cost vs. Number of Fragments

Figure 6(a): Plot of Min Total IO Cost vs. Number of Fragments for Part 3 to find the Optimal Emp class partition scheme

Scatter Plot of Total IO Cost vs. Number of Fragment

Figure 6(b): Scatter Plot of Total IO Cost vs. Number of Fragments for Part 3