A Cost Model for Clustered Object-Oriented Databases*

Georges Gardarin[†◇]
[◇]Prism Laboratory
University of Versailles
78035 Versailles, France
last-name@prism.uvsq.fr

Jean-Robert Gruser[†]
[†]Projet Rodin
INRIA, Rocquencourt
78153 Le Chesnay, France
last-name@rodin.inria.fr

Zhao-Hui Tang[◇‡]
[‡]TechGnosis France
34, Bld Sebastopol
75003 Paris, France

Abstract

Query processing is one of the most critical issues in Object-Oriented DBMSs. Extensible optimizers with efficient search strategies require a cost model to select the most efficient execution plans. In this paper we propose and partially validate a generic cost-model for Object-Oriented DBMSs. The storage model and its access methods support clustered and nested collections, links, and path indexes. Queries may involve complex predicates with qualified path expressions. We propose a method for estimating the number of block accesses to clustered collections and a parameterized execution model for evaluating predicates. We estimate the costs of path expression traversals in different cases of physical clustering of the supporting collections. The model is validated through experiments with the O2 DBMS.

1 Introduction

One of the basic functionalities of object-oriented database systems is a declarative query language derived from an object extension of SQL. Standard query languages have been proposed [Cat93, Mel93] and the issue of query optimization is now crucial. Research has been done on query rewriting techniques [GM93, HCF+89, CD92, MDZ93, FG94] and search strategies for discovering optimal execution plans [IK90, LV91]. All these techniques assume a cost model to evaluate query plans, taking into account the I/O and CPU costs. Little work has been done to define and validate a generic cost model for object databases. Cost models have been sketched for several optimizers [BF92, BMG93, CD92, COA+94, Zai94], but to our knowledge, none of them have been implemented and validated on real databases.

In this paper, we assume that the query optimizer is able to break the encapsulation of objects and look at the data structure used to implement them. We then propose a generic cost model for object databases, which takes into account various aspects of object storage models including:

- Clustering. Object of different classes can be physically grouped together according to multiple predicates with a priority scheme.
- Linking and Embedding. Associated objects can be linked through object identifiers; nesting target objects of associations inside source objects to constitute composite objects is also supported as a variant of clustering.
- Indexing. Simple indexes on attributes of a class and complex indexes on path expressions are supported.
- Methods. The cost of user operations on objects are integrated in the model through parameterized access to computed attributes.

This generic cost model has been validated on simple queries using the O2 database system. The results show good predictions for searches of clustered collections.

The main contribution of our approach is first, the generality of the cost model. It is defined by a

*This work has been partially funded by the Commission of the European Communities under the Esprit project IDEA.

**Proceedings of the 21st VLDB Conference
Zürich, Switzerland, 1995**

class graph with clustering predicates, embedded and linked classes, path indexes, and virtual attribute access costs. Thus, the generality of the cost model is assured by both its generality and the introduction of relevant parameters. Second, our cost formulas include an evaluation of the block hit rate when evaluating a predicate on a clustered collection: we extend the Yao formula [Yao77] to the clustered case. Third, we estimate the cost of searches with complex predicates to include qualified path expressions [JWKL90]. Finally, our cost model is validated through experiments on the O2 DBMS. The validation demonstrates the model validity for processing queries.

The organization of this paper is as follows. In section 2 we present the physical storage model, particularly how objects from different classes are clustered or embedded together and how path indexes are integrated. Section 3 is devoted to a discussion of the execution model. We focus on the evaluation of predicates with path expressions. In section 4 we first introduce the cost model parameters and then present a method for estimating the number of block accesses to a clustered collection given the predicate selectivity. The next section describes the cost formulas for searching through collections with complicated predicates. Section 6 uses these results to compute the costs of sequential and index scans. Section 7 describes the results of experiments done on the O2 DBMS to validate our analytical cost model.

2 The Object Storage Model

In this section, we introduce a generic storage model for Object-Oriented DBMSs. This model captures various types of object identifiers, clustering techniques, and indexes. It is general enough to be specialized for representing the internal model of most object-oriented DBMSs, as shown below with O2.

2.1 Object Identifiers

Most object-oriented database systems store object on slotted pages. A slotted page contains a variable array of slots at the end of the page. The offset of each object from the beginning of the page is kept in the slot. Physical object identifiers (OIDs) consist of the segment number, the page number and the slot number. Physical OIDs are used in ObjectStore for example [Obj94] [GA93]. Although physical OIDs are efficiently decoded, they do not allow free movement of objects in databases. To avoid this problem, other systems have implemented logical OIDs, a value that maps through a hash table to retrieve the actual location of the object. Our generic cost model considers the type of OID as an input parameter. This parameter effects directly the cost of *object link* traversals.

2.2 Indexing

To speed up predicate evaluation, Object-Oriented DBMSs support classical indexes to class instances. Indexes are generally organized as B-trees containing attribute values with OID lists. Indexes can be clustered. In our model, an index is a function from an attribute value to a list of OIDs. The cost of traversing the index is a parameter giving the cost formulas of the function.

Further, some Object-Oriented DBMS provide path indexes, which follow a sequence of objects linked by relationships. There exist various implementation of path indexes [BK89]. We consider a path index as a function associating the end of the path to all prefixes of the path. The cost of the function can be changed to model various implementation.

2.3 Object linking and Embedding

In most Object-Oriented DBMSs internal models, associated objects can be stored together as composite objects or stored separately and linked through OIDs. In the case of composite objects, a 1-1 or 1-N relationship gives rise to an object with 1 or N embedded objects. In the case of linking, a relationship gives rise to a mono or multi-valued attribute pointer. Linking or embedding related objects is a very important decision for query costs. Thus, our model captures this structure through a graph description. Links are represented by edges between their class nodes. Embeddings are also represented by links. We distinguish embedding links by marking them using a double-arrow-dotted line in the database schema and placement graph.

2.4 Clustering

Database accesses in an Object-Oriented DBMS are much more complex due to the rich variety of type constructors provided. In addition to traditional scans of sets of objects, Object-Oriented DBMSs often use navigation-like access among related objects. Such inter-object references can generate random disk access if the entire database does not fit in main memory. A well-known approach to speed access to related objects is clustering by object associations. Clustering groups objects of possibly different classes into contiguous disk pages (i.e., clusters) using information on object contents or relationships (i.e., association, inheritance, aggregation). To capture a large set of clustering strategies, we adopt a predicate based definition of clustering. Predicates that reference class properties are used to define the set of objects stored together in clusters. To support shared objects, we adopt the priority concept described in [GA93]. Clustering information is presented using a graph defined by the

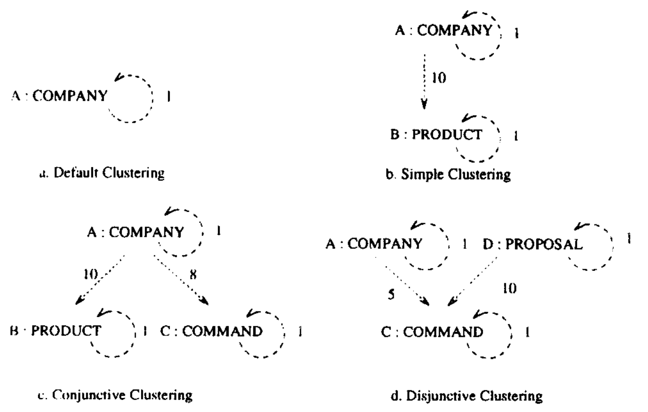


Figure 1: Clustering Possibilities

database administrator. Edges correspond to clustering predicates. A priority weight from 1 to 10 is given to each edge. For object instances of nodes shared by several edges, the edge with the highest weight is selected.

Figure 1 shows four possibilities for collection clustering as follows:

The default clustering : All the objects of the collection are physically grouped in contiguous disk space. If no clustering predicate is provided for a collection, this default grouping is implicit. In Figure 1.a, *company* has the default clustering.

The simple clustering : This is the classical clustering of two collections according to a join predicate. In Figure 1.b, the *product* collection objects are clustered with the corresponding objects of the *company* collection with priority 10. All the companies which do not have any products are grouped together with priority 1. All products of unknown company are grouped together with priority 1.

The conjunctive clustering : In Figure 1.c, all the *product* objects and the *command* objects are clustered with their associated companies. This clustering strategy allows the clustering of several collections with one.

The disjunctive clustering : In the case of Figure 1.d, each *command* object should be stored with either the associated *company* or the associated *proposal*. Since objects cannot be duplicated, they are assumed to follow the edge with the highest weight.

2.5 A Database Example

Figure 2 shows an example of a database relationship and its physical data grouping. Solid lines with arrows represent the database schema. The cardinality permitted by the relationship type is indicated by the arrows at the ends of the solid line. For example, the line between *company* and *command* indicates a one-to-many relationship between these two collections. A path index is represented by a set of arrow lines starting from the collection at the end of the path and pointing to all the intermediate collections. For example, *Company.Command.Proposal* is a path index. Dotted lines indicate object clustering, they are labelled by priority weights. A double-arrow dotted line represents object embedding. The label of a dotted line can include a predicate for clustering. In the graph, all the products produced after 1994 are clustered with their company with priority 3. Since there is no dotted line between *command* and *company*, *commands* are stored independently of their *companies*.

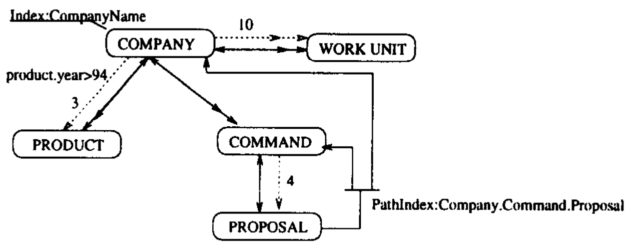


Figure 2: Placement Graph

```
cluster COMPANY on (workunits, products);
cluster COMMAND on (proposition);
```

Figure 3: Creating a cluster in O2

The placement graph induces a physical organization of data. Each generated cluster can be seen as a partition of a collection. For example there are two partitions for *company* : $Cl_{(company,workunit)}$ (*work unit* objects are embedded with *company* objects) and $Cl_{(company,workunit)\rightarrow product}$.

2.6 Object Clustering in O2

A typical O2 application runs in client-server mode. Clients and servers exchange physical disk pages. Thus data transfer is quite expensive and data clustering can largely improve performance since it decreases communication cost. In O2, objects are mapped into page slots and a physical cluster is a set of records [BD90]. Logical cluster in O2 has no size limit, whilst a physical cluster is bounded to the page size. The clustering strategy is defined using a placement tree. When instances of a given class are created, the placement tree associated with this class generates the physical organization on secondary memory.

3 An Execution Model for Predicate Evaluation

The execution model for scanning a collection with predicate evaluation uses an algorithm that involves three low level operators : Fetch, Comp (for compare two values) and Dot (for get an attribute of an object

.Dot
Retrieve the value of a tuple field or apply a method on the object state with the given arguments. The state of the object must be already present in memory. It returns the value of the attribute or the resullt of the method.

.Comp
Evaluate a simple predicate and returns True or False.

Fetch
Find the physical address of an object by its OID and loads it into memory if it is not charged yet.

Figure 4: Low Level Operators

or apply a method). The combination of these low level operators directly influences the cost of a query execution. In this section, we detail the three low level operators and discuss the algorithm for evaluating predicate with path expression.

3.1 Low Level Operators

Low level operators are described in Figure 4. We distinguish the access to an object from its OID (Fetch operator), the evaluation of a simple predicate on an object already in memory (Comp operator), and the projection of an object on an attribute or method value (Dot operator parameterized by the attribute or method name). The costs of these three operators are parameters of our cost model.

In general, Fetch costs little CPU time but one or zero I/Os; on the contrary, Comp and Dot cost only CPU time. The Fetch operator should be applied before evaluating a predicate or before getting an attribute value, especially when traversing a collection with a qualifying predicate. The Dot operator is generic in the sense that it is parameterized by a property name. The property can be an attribute or a method; the property is an important parameter for cost computation.

3.2 Path Expression Evaluation

An atomic predicate can be a simple predicate (e.g., x in Product, x.year > 30) or a complex predicate containing a path expression. Our model integrates qualified path expressions. Thus, each collection involved in a path expression can be qualified with a simple predicate [FLU94]. For example, let x be a variable on collection *Person*

```
x.vehicle[color="Red"].company[name="Renault"];
```

is a valid path expression. A query with a predicate containing a path expression can be executed in different ways. At the execution level, to estimate the

Given a sub-path expression
$(x_i \in C_i, x_i(P_i).c_{i+1}(P_{i+1})..c_n(P_n).attribute)$
where c_i is a set of references to an object of C_i and P_i is a qualified predicate, we have :

Depth_First_Fetch Algorithm:

```
DFF(xi(Pi).ci+1(Pi+1). .. .cn(Pn))
  {
  if (i<n)
   {
   for x in (xi.ci+1)
    {
    FETCH(x)
     if COMP(Pi) = TRUE
       return DFF((FETCH(x.ci+1)(Pi+1)
             .ci+2(Pi+2). .. .cn(Pn)))
    }
   }
   else return COMP(Pn)
  }
```

Figure 5: The DFF Algorithm

cost of a path traversal, we consider a Depth-First-Fetch (DFF) evaluation given in Figure 5. We select this algorithm because it is implemented in DBMSs such as O2 and appears to be the best algorithm with large memory. It recursively processes the object composition graph corresponding to the path using depth first search. (Breadth first search is also possible, but the path must be rewritten into equivalent logical algebraic expression [CD92] and expressed as a sequence of explicit joins in the execution plan. The cost model will then compute the cost of the cascaded joins, using traditional join cost formulas [Zai94]). After the query optimization procedure, if an execution plan still contains a path expression, we assume that the optimizer specifies the Depth-First-Fetch method.

When using the DFF procedure, we evaluate predicates of intermediate collections along the path expression once the instances of the intermediate collections are loaded into memory. The advantage of this approach compared to traditional depth-first-fetch is that it avoids joining intermediate results several times. For example, if a query has to find the person who has a red car made by company Renault, i.e., x.car.company[name="Renault"] and x.car[color = "red"], qualified path expressions only traverse the x.car collection once. More generally, we permit any predicate that is a conjunction of disjunctions of atomic predicates.

In summary, we assume a rather specific execution model, but with sufficient parameterization (e.g., for low level operators) and flexibility (e.g., for path expressions) to capture a large class of search algorithms.

We also provide several join cost formulas not described in this paper due to space limitation. Further, we use clustering, embedding, linking, and indexing information to compute the number of objects processed by a query.

4 Cost Model Parameters and Clustered Block Hit

A cost model is a set of formulas to estimate the cost of an execution plan. Cost-based query optimizers select the most efficient execution plan based on the cost estimations among the equivalent execution plans [Zai94]. There are several major components of the cost: CPU_Cost, IO_Cost, COM_Cost. CPU_Cost is the cost of executing CPU instructions, for example, evaluating a predicate, executing a loop. IO_Cost is the input/output cost for reading and writing data between memory and disk. COM_Cost is the cost of network communication among different nodes. In this section, we present a cost model designed for an object oriented database which permits object clustering. In our case, since the database is centralized, our cost model focuses on the IO_Cost and CPU_Cost; the communication cost is not taken into account. A cost model can be very complex. We aim at defining a usable cost model for a query optimizer. Thus, it has to remain as simple as possible. To simplify the evaluation of cost estimations, we assume that objects have a size less than a page, also, we use a uniform distribution model for attribute values in domains.

4.1 Parameters

The cost model uses statistics about the database components to estimate a given execution plan cost. The statistics contain information concerning collections in the database like the cardinality of collection, the size of objects, the number of distinct values of an attribute, indexes and clustering. The system parameters and some calculated expressions are also included in these statistics. This section describes the details of these parameters.

Statistical information on collections are defined as follows:

- $||C||$: cardinality of collection C
- $||C_i||$: cardinality of cluster i of collection C
- $|C_i|$: number of pages of cluster i of collection C
- S_C: average object size in collection C

Statistical information on attribute distribution are as follows:

- fan_{C_1,C_2}: average number of references from a C1 object to C2 objects
- D_{C_1,C_2}: number of distinct references from a C1 object to C2 objects.
- X_{C_1,C_2}: number of C1 objects having NULL references to C2 objects.

From these parameters, we calculate the following expressions :

- Z_{C_1,C_2}: average number of distinct references to C2 objects of those C1 objects who have at least one non NULL reference:

$$Z_{C_1,C_2} = \frac{D_{C_1,C_2} * ||C_1||}{||C_1|| - X_{C_1,C_2}}$$

- Sel(Predicate): The selectivity of a predicate. (usual formulas can be found in [SAC+79, MCS88, PSC84]).

Others parameters:

- b: B tree fanout
- BLevel(I): number of Btree levels of the index I
- S_p: the page size
- Proj_Cost: CPU cost to project one attribute of an object
- m: available memory (unit: page)
- Cost_load_page : Cost to load a page in memory (one IO)

4.2 Estimating the Page Number of a Clustered Collection

In this section, we present a method to estimate the page numbers of each partition of a collection when it is clustered with other collections.

In a storage system where there is no object clustering (like in the default clustering of Figure 1.a), there are $\lfloor \frac{s_p}{s_A} \rfloor$ objects in a page and the total pages of a collection can be calculated by:

$$|A| = \frac{||A||}{\lfloor \frac{s_p}{s_A} \rfloor}$$

But in a system where objects of different collections can be clustered together, the page number can not be calculated using the above formula, since a collection may have several physical partitions. In a given partition, objects belong to the same collection while in other partitions objects are physically grouped with objects from different collections. We propose a method for estimating the total pages of a collection after clustering. We assume that objects are not duplicated and the cardinality of a collection is not changed after clustering. Suppose that collection B is clustered with collection A (see Figure 1.b). The average object size of collection A objects is $S_A$and the average size of collection B objects is S_B. We suppose that S_A and S_B are both smaller than a page size.

We first estimate the number of pages of collection A, which is the root of the cluster tree. There are two physical partitions of collection A after clustering: a partition clustered with B $Cl_{A\rightarrow B}$ and a partition Cl_A where there are only collection A objects inside.

- For Cl_A, the number of collection A objects inside is $||A_{Cl_A}|| = X_{A,B}$, then :

$$|A|_{Cl_A} = \frac{X_{A,B}}{\lfloor \frac{S_p}{S_A} \rfloor}$$

- For $Cl_{A\rightarrow B}$, the number of root objects in this cluster is $||A|| - X_{A,B}$, the cluster size is $S_{Cl_{A\rightarrow B}} = S_A + (Z_{A,B} * S_B)$, then

$$|A|_{Cl_{A\rightarrow B}} = \begin{cases} \frac{||A|| - X_{A,B}}{\lfloor \frac{S_p}{S_{Cl_{A\rightarrow B}}} \rfloor} & \text{if } S_{Cl_{A\rightarrow B}} < S_p \\ ||A|| - X_{A,B} & \text{if } S_{Cl_{A\rightarrow B}} \geq S_p \end{cases}$$

The total number of pages is

$$|A| = |A|_{Cl_A} + |A|_{Cl_{A\rightarrow B}}$$

In the second step, we estimate the page numbers of collection B (a non-root node in clustering graph) after clustering. The different partitions where we can find of collection B are $Cl_{A\rightarrow B}$ and Cl_B.

- For Cl_B, the number of B objects which is not referred by any A objects is $||B|| - (||A|| * D_{A,B})$, then we have

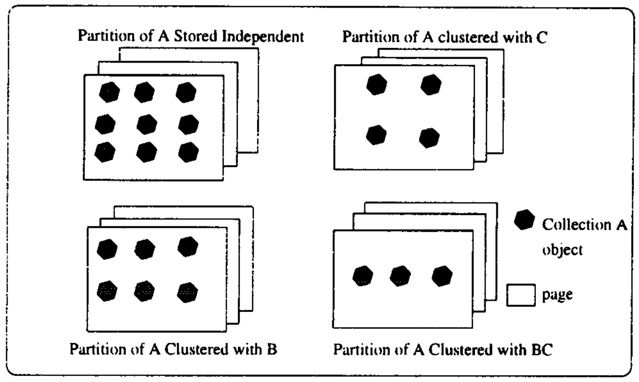
$$|B|_{Cl_B} = \frac{||B|| - (||A|| * D_{A,B})}{\lfloor \frac{S_p}{S_B} \rfloor}$$

- For $Cl_{A\rightarrow B}$ the size and the number of objects per *Cluster* remains the same, but the number of *Clusters* to access is now $X'_{Cl_{A\rightarrow B}} = Min(Z_{A,B} * X_{A,B}, X_{A,B})$ and we have :

$$|B|_{Cl_{A\rightarrow B}} = \begin{cases} |A|_{Cl_{A\rightarrow B}} & \text{if } S_{Cl_{A\rightarrow B}} < S_p \\ \text{else} & \\ ||A|| - X_{A,B} & \text{if } Z_{A,B} * S_B \leq S_p \\ (||A|| - X_{A,B}) * \lfloor \frac{Z_{A,B} * S_B}{S_p} \rfloor & \text{if } Z_{A,B} * S_B > S_p \end{cases}$$

And the total number of pages is

$$|B| = |B|_{Cl_B} + |B|_{Cl_{A\rightarrow B}}$$



Partition of A Stored Independent
Partition of A clustered with C
Collection A object
page
Partition of A Clustered with B
Partition of A Clustered with BC

Figure 6: Physical partitions of collection A after clustering

We use the same method of calculation for the conjunctive, the disjunctive and the combination of all clustering techniques. Thus, in the case of more complicated clustering, the complexity of the formulas increases with the shape of the placement tree.

4.3 Extending the Yao Formula to Clustered Collections

During query optimization, the optimizer computes the selectivity of a predicate and we need to estimate the number of page accesses to a collection. In relational database, Yao [Yao77] has derived a formula to calculate the expected number of page access. This is given by the following theorem:

Theorem Yao: Given n records uniformly distributed into m blocks(1 < m <= n), each contains n/m records. If k records (k <= n) are randomly selected from the n records, the expected number of block hits is given by

$$yao(n, m, k) = m * [1 - \prod_{i=1}^{k} \frac{nd - i + 1}{n - i + 1}] \text{ where d = 1 - 1/m}$$

For a clustered collection C , we can not simply apply Yao's formula to estimate the page hits by $yao(||C||, |C|, Sel(P) * ||C||)$ as often done in RDBMS. The reason is that a clustered collection has more than one partition. Certain objects are clustered with objects from different collections, while others are stored alone. Thus, the densities of objects in the different partitions are not the same. In this case, we extend the Yao formula to Yao' given by the following theorem:

Theorem yao': Given a collection C which has p partitions, each partition has $||C||_i$ objects. If k ($k < \sum_{i=1}^{p} ||C_i||$) objects are randomly selected from C, the expected number of block hits is given by

$$yao'(C, k) = \sum_{i=1}^{p} yao(||C_i||, |C_i|, k_i)$$

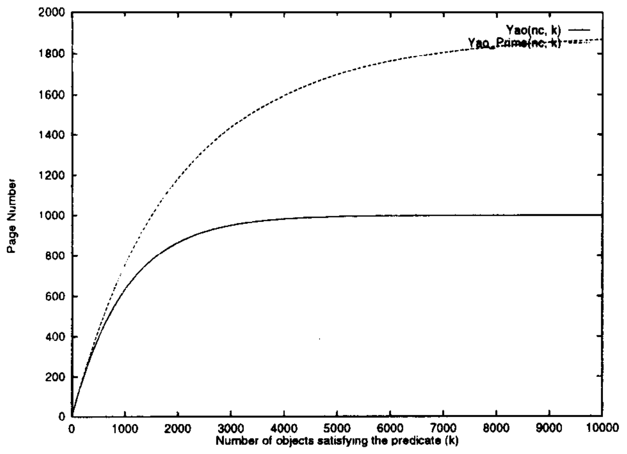


Figure 7: comparison of formule Yao and Yao'

where k_i is the number of objects to be selected in partition C_i.

If objects to be selected are placed uniformly among all the partitions, we have $k_i = (||C_i||/||C||) * k$. When the objects that satisfy the predicate are not placed uniformly among the different partitions, we have to use the selectivity on each partition to get the right value of each k_i. The yao$'$ formula is more general than the Yao formula, and the latter is a particular case of yao$'$ when collection C has only one partition.

We now illustrate the two formulas through a simple example. Suppose we have 3 collections (see Figure 6): A,B, and C where B and C are clustered with A. The average object sizes of collection A, B, C are the same, which equal to one-tenth of a page. The cardinality of A is 10,000. After clustering, 1500 objects of collection A are stored with B and C, 3000 A objects are stored with B, 2500 A objects are grouped with C and the remaining 3000 A objects are stored independently. We select the object A with a given predicate.

In Figure 7, we trace the results of the Yao and Yao' functions for different predicate selectivities. We suppose the selectivity is uniform among all the clusters. The vertical axis represents the number of pages of collection A to be accessed and the horizontal axis is the number of objects that satisfy the selection predicate. When the selectivity is greater than 10 percent, we notice a significant difference between Yao and Yao'. The value of Yao is lower than Yao' since it considers that all the C objects are grouped together, thus the density is higher and the probability of two objects stored in the same page is higher. This result verifies that when objects are clustered with other collection objects, the sequential scan and index scan become costly.

5 Cost Formulas for Typical Operators

5.1 Cost of Dot

The Dot operator is applied to retrieve the value of an attribute when the object is present in memory or to execute a method on an object state. The attribute can be calculated through a method. Thus, to evaluate the CPU cost of Dot, a user given parameter is required for non directly implemented attributes. We assume that objects document themselves through methods. Then, we have :

IO_Dot_Cost = 0
CPU_Dot_Cost = constant or the CPU cost of the method

5.2 Cost of Comp

The Comp operator is applied to test a predicate when the object is present in memory. Before applying the Comp operator, we have to apply the Dot operator to compute the attribute values in memory. Thus, we have :

IO_Comp_Cost = 0
CPU_Comp_Cost = constant (If no predicate is defined, the default predicate is set to TRUE and, in this case, we have : CPU_Comp_Cost = 0)

5.3 Cost of Fetch

The cost of fetch depends on whether the object is present in memory or not :

$$\text{IO\_Fetch\_Cost} = \begin{cases} 0 & \text{if the object is in memory} \\ 1 & \text{if a page has to be loaded} \end{cases}$$

CPU_Fetch_Cost = 0

5.4 Cost of Accessing an Index

We assume that indexes are implemented with B tree structures as usual. Suppose the Btree fanout is b, with Blevel levels, the CPU cost of index scan is

$$\begin{aligned} CPU\_index\_Cost(I) &= log_2 b(I) * Blevel(I) \\ &\quad *CPU\_Comp\_Cost \end{aligned}$$

The IO cost of accessing an index is equal to the index tree level. IO_index_Cost(I) = Blevel(I)

5.5 Cost of Creating Temporary Results

During the execution of a query, the system needs to generate some temporary results. The size of the results depends on the predicate applied on the input collection C and on the projected attribute length

(S_{proj}). We determine the size of the output collection by :

$$\begin{aligned} ||out|| &= Sel(P) * ||C|| \\ |out| &= \frac{||out||}{\lfloor \frac{S_p}{S_{proj}} \rfloor} \end{aligned}$$

Managing the output collection requires IOs only when the results have to be swapped on disk; thus, if m is the number of pages available in main memory, we get:

$$\text{IO\_out\_Cost(P,C,proj)} = \begin{cases} |out| - m & \text{if } m < |out| \\ 0 & \text{otherwise} \end{cases}$$

The CPU cost is the time to construct the $||out||$ objects given by:

$$\text{CPU\_out\_Cost(P,C,proj)} = ||out|| \text{ *Proj\_Cost*} Nb_{proj}$$

5.6 Cost for Evaluating Predicate

We assume that the required pages of the input collections are already loaded in memory. Two cases have to be considered :

- predicate without path expression
- predicate with path expression

5.6.1 Predicate without Path Expression

The CPU_Cost is the cpu time for making a Dot on each objects and for comparing two atomic values. The IO_cost is null since the object state is already in memory. It yields :

$$\begin{aligned} IO\_Eval\_Cost(P) &= 0 \\ CPU\_Eval\_Cost(P) &= ||C|| * (CPU\_Dot\_Cost \\ &\quad +CPU\_Comp\_Cost) \end{aligned}$$

5.6.2 Predicate with Path Expression

In the execution model, a predicate with path expression is executed with Depth_First_Fetch algorithm. Of course there are other ways of evaluating a path expression as explicit joins [GGT95]. These execution plans are expressed without path expression at the physical level. The Depth_First_Fetch has to find for each object of the input collection, the corresponding attribute.

Since they are p memory pages to process the predicate, we make the assumption that the number of pages is at least equal to the path length ($p \geq n$), which is usually the case. Thus there is at least one page for each collection in memory during execution. Given a path expression ($x \in C_1, x.c_2(P_2).c_3(P_3)..c_n(P_n).attribute$) where c_i is a reference to an object of C_i and S_i are the selectivity of the predicate P_i ($S_i = Sel(P_i)$).

We first define the number of distinct references involved in the Path $Ref_i = (1 - Prob_i) * ||C_i||$. P_i is the probability of an object of collection i to be not involved in the path, we have :
$Prob_i = (1 - \frac{1}{||C_i||})^{(Ref_{i-1} * S_{i-1} * fan_{C_{i-1}, C_i})}$
with $Ref_1 = ||C_1||$.

Depending on the memory size, we have three hypotheses :

- if p is large enough to store all pages involved in the path expression, each page is loaded once and only once. The Yao' formula approximates the number of pages to be loaded : this is the large memory hypothesis.

$$Page_{min} = \sum_{i=2}^{n} \begin{cases} 0 \text{ if } C_{i-1} \text{ is clustered with } C_i \\ Yao'(C_i, Ref_i) \end{cases}$$

- if p is equal to the path length (p = n), we have the extreme case of the small memory hypothesis.

$$NbPage_{max} = \sum_{i=2}^{n} \begin{cases} 0 \text{ if } C_{i-1} \text{ is clustered with } C_i \\ ||C_1|| * \prod_{j=2}^{i} fan_{C_{j-1}j, C_j} * S_{j-1} \end{cases}$$

- if $n < p < Page_{min}$, the memory is not able to store entirely all involved pages of the path expression. The IO cost will directly rely on the page replacement policy mechanism of the database. Some pages will be reloaded many times. In this case, we approximate the IO cost by a linear function on linking the maximum and the minimum number of pages. We assume that such an approximation is sufficient to compare execution plans. We have :

$$\begin{aligned} NbPage &= \frac{Page_{max} - Page_{min}}{Page_{min} - n} * (n - p) \\ &\quad +Page_{max} \end{aligned}$$

Finally the IO cost of a predicate P evaluation with p pages is :

$$IO\_Eval\_Cost(P) = \begin{cases} NbPage & \text{if } n \leq p < Page_{min} \\ Page_{min} & \text{if } p \geq Page_{min} \end{cases}$$

The discrepancy of execution plans is based on the comparison between break points of the search cost curves.(see Section 7) The break point is given by : $BreakPoint = Page_{min} * S_p$.

The CPU cost of this evaluation is the cost of doing a Dot along all objects and a Comp for all simple predicate in the path :

$$\begin{aligned} CPU\_Eval\_Cost(P) &= ||C1|| * (CPU\_Dot\_Cost \\ &+ CPU\_Comp\_Cost) \\ &*(\sum_{i=2}^{n} \prod_{j=2}^{i} fan_{C_{j-1},C_j} * S_{j-1}) \end{aligned}$$

6 Search Cost

We define *Search* the operation which applies a predicate to a possibly indexed collection and retrieves the results. The cost of the search operation can be broken up into several components : the cost of accessing an index (if it exists), the cost of loading the collection, the cost of evaluating the predicate, and finally the cost of building the result.

The IO cost of a search is the cost of loading the relevant pages of the collection C plus the IO cost of the predicate evaluation. Two cases are considered, sequential scan and index scan.

6.1 Sequential Scan

The basic operation formulas given in the previous section entail:

$$\begin{aligned} IO\_scan(C, P, proj) &= |C| \\ &+ IO\_Eval\_Cost(P) \\ &+ IO\_out\_Cost(P, C, proj) \end{aligned}$$

The CPU cost is proportional to the number of objects to scan :

$$\begin{aligned} CPU\_scan(C, P, proj) &= CPU\_Eval\_Cost(P) \\ &+ CPU\_out\_Cost(P, C, proj) \end{aligned}$$

6.2 Index Scan

If an index I is used to access the collection, the IO cost only takes into account the number of pages of C determined by accessing the index. We determine the restricted collection C' by :

$$|C'| = \begin{cases} yao'(C, ||C|| * Sel(P)) & \text{if I is non-clustered} \\ Sel(P) * |C| & \text{if I is clustered} \end{cases}$$

$$||C'|| = Sel(P) * ||C||$$

It follows that :

$$\begin{aligned} IO\_ind\_scan(C, P, proj) &= IO\_index\_cost(I) \\ &+ |C'| + IO\_out\_Cost(P, C, proj) \end{aligned}$$

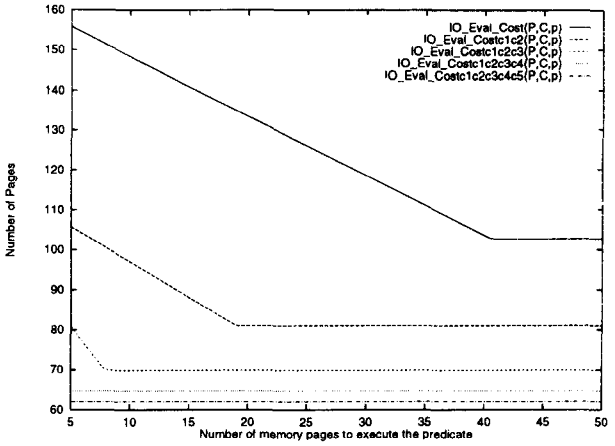


Figure 8: IO cost of scan

The CPU cost derives from the evaluation of a predicate P over the objects of C :

$$\begin{aligned} CPU\_ind\_scan(C, P, proj) &= CPU\_index\_cost(I) \\ &+ CPU\_out\_Cost(P, C, proj) \end{aligned}$$

Finally, the total time to evaluate a search operation is obtained by converting the IO_Cost to the unit of time by multiplying the time that takes one IO operation. That yields :

$$\begin{aligned} Total\_time &= IO(\_ind)\_scan * Cost\_load\_page \\ &+ CPU(\_ind)\_scan \end{aligned}$$

To illustrate the formulas, we use our model to compute the IO cost of a sequential search in function of the memory buffer size. The results are given in Figure 8. The curves represent the number of IOs for scanning a collection of 1000 objects clustered with objects from different collections. The predicate of scan contains a path expression over 5 collections ($x \in C_1, x.c_2.c_3.c_4.c_5.attribute = value$). The number of objects pointed in the path expression is half of the number of objects of the precedent collection ($D_{C_i,C_j} = 0.5$). There are five curves in the figure, which represent the different cases of physical clustering of objects. It is clear from our result that when collections are clustered together, the IO cost for evaluating a predicate with a path expression is lower.

7 Cost Model Validation

The validation of our cost model is done on the O2 Client/Server database system [BCD89] on a SPARC station LX. The O2 server and the O2 client run on the same machine on a single user configuration. The cost of an IO is the time to load a page of 4 kBytes into the buffer of the server plus the time to transmit this page to the buffer of the client. We measure an average time of 25 ms to load a page and transmit it.

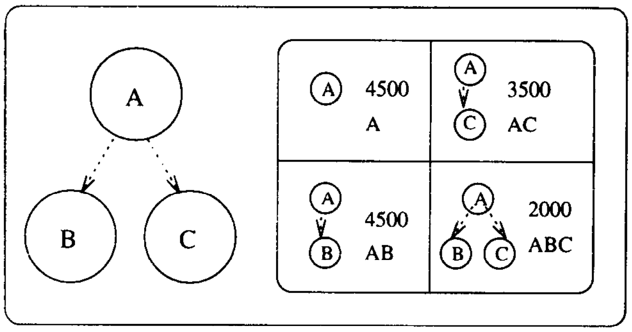


Figure 9: Example of a Database

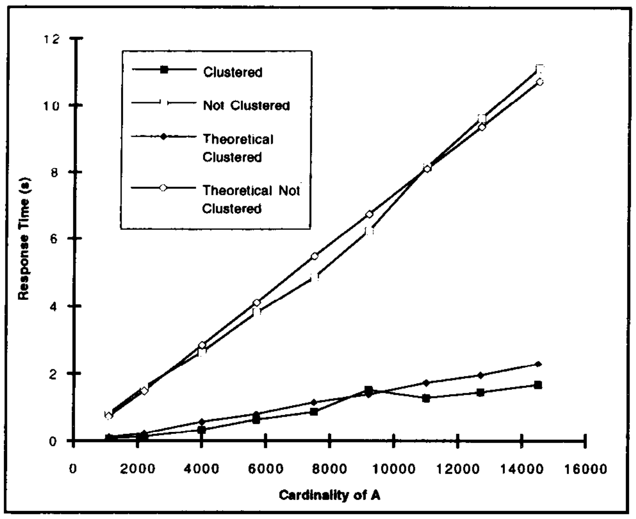


Figure 10: Evaluation of Path Expression

Queries are executed on two equivalent databases: the first one is clustered in the way described in Figure 9 and the second one uses the default clustering strategy for each collection. We create an index on the attribute "IndexId" of each collection. The links between A, B and C are generated at random. A non indexed attribute "id" gives a unique identifier for each object. During our experimentation, each query is executed 10 times in the same condition. The average value of response times is used as an element of the result.

Clustering Validation :

In the first experiment, we process the following query on these two databases, which scans A collection with a predicate containing a path expression traversing B and C collections.

```
select x from x in A where x.b.c.id != 0 ;
```

The experiment validates how the clustering influences the execution cost of this query and how the results match our cost model. We measure the response times by varying the cardinality of collection A (see Figure 10). The result shows how the clustering technique improves the performance of the query. In this example, the performance of the clustered database appears six times better than a non clustered one which corresponds to the estimations given by the formulas.

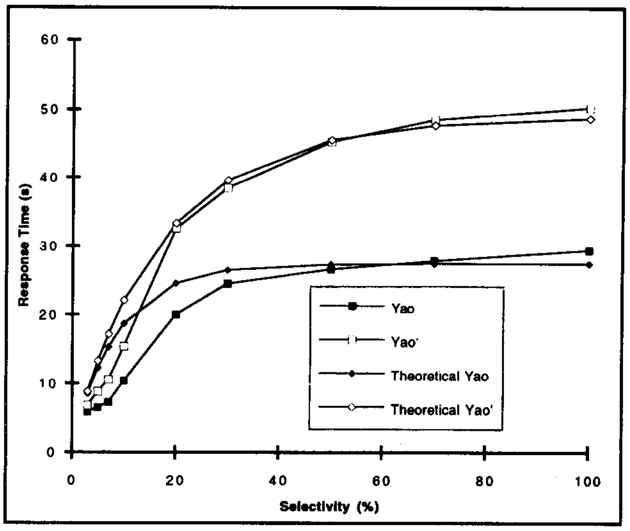


Figure 11: Yao' Validation

Yao' Validation :

Experiments have been done to validate the Yao' formula. We process the following query on the two separate databases of Figure 9.

```
select x from x in A where x.IndexId >= N% ;
```

The query scans the collection A by using an index on IndexId. We use an index to avoid scanning the whole collection. The non clustered database verifies the usual Yao formula while the clustered database validates the Yao' formula.

The size of an object is 300 bytes and a data page can hold 12 objects. Figure 11 shows the results with multiple selectivities in the case of a large memory size. The proportionality between the two formulas is respected by our cost model. But for low selectivity, there is a difference between the experimental curves and the theoretical ones. We explain these differences by the effect of the UNIX buffer mechanism for small collections. When the client queries the server for pages, the server may use pages already present into memory; this mechanism is more sensible for a small number of pages.

Search Validation :

In this experiment we want to show how the memory size impacts the execution of a search when collections are clustered. We execute the query of the first experiment on these two databases and we vary the memory

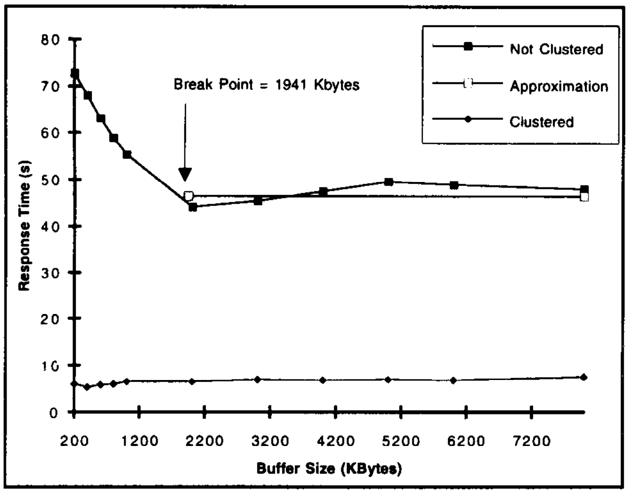


Figure 12: Search Execution with Clustering

buffer size for processing the query. To reduce the effect of the UNIX buffer, we flush out the system buffer before each query. Figure 12 shows time spent for executing the search with different memory sizes.

The first remark comes from the difference between the clustering and the non clustering case : clustered collections (due to the number of IO) much better tolerate memory size variations.

In the case of default clustering of these three collections, we have : $Ref_1 = 14500$, $Ref_2 = 3965$ and $Ref_3 = 1202$. Then $Page_{min} = Yao'(C2, Ref_2) + Yao'(C3, Ref_3) = 485.26$. The Break Point is equal to $485.26 * S_p \simeq 1941 KBytes$. It is important to notice that the model allows us to determine the break point of the curve. When memory size is smaller than this point, the IO cost becomes much higher for depth-first-fetch algorithm and the query optimizer should make a choice between pointer navigations and a set of binary joins.

Errors Analysis:

Figure 13 summarizes for all experiments the maximum error by overestimation (Max-O) and the maximum error by underestimation (Max-U), the average overestimation error (Ave-O) and the average underestimation (Ave-U), with the standard deviation (St-Dev). The corresponding formulas can be found in [Knu68, Swa89, Zai94]. We notice that, for these simple queries, the percentage of errors are low. But for more complex queries with nested predicates, the error rate may increase due to the complication of the cost formulas.

8 Conclusions and Future Work

This paper has presented a cost model for query optimizer in an Object-Oriented system. The model takes into account object clustering and indexing. Based on the statistics and placement information, the model

Experiments/Errors	Max-O	Max-U	Ave-O	Ave-U	St-Dev
Clustered	0,094	0,725	0,094	0,091	0,328
Not Clustered	0,099	0,129	0,020	0,016	0,149
Yao	0	1,915	0	0,192	4,206
Yao'	0	0,648	0	0,065	2,067
Search	0,064	0,143	0,016	0,014	0,576

Figure 13: Errors

correctly estimates the cost of different access methods for scanning collections and evaluating predicates with path expressions. We also propose a method to estimate page accesses to a clustered collection (Yao'). Further, we explore the effect on the cost of query execution of different object grouping cases. Experiment with O2 system allow us to verify the most important hypothesis of the proposed cost model.

In the next future, we plan to do more performance tests to validate all the cost formulas. We will like to compare the costs of evaluating path expressions using explicit joins versus OID pointer navigations. Results would be helpful to find good heuristic strategies for Object-Oriented query optimizers. We plan to extend our cost model to a distributed system where communication costs will be considered.

References

[BCD89] F. Bancilhon, S. Cluet, and C. Delobel. A query language for the O2 object-oriented database system. In R. Hull, R. Morrison, and D. Stemple, editor, *Proc. 2nd Intl. Workshop on Database Programming Languages*, page 122, Gleneden Beach, Oregon, June 1989. Morgan Kaufmann.

[BD90] V. Benzaken and C. Delobel. Enhancing performance in a persistent object store: Clustering strategies in O2. In *Fourth Int'l Workshop on Persistent Object Sys.*, page 375, Martha's Vineyard, MA, September 1990.

[BF92] E. Bertino and P. Foscoli. An analytical model of object-oriented query costs. In *Persistent Object Systems, Workshop in Computing Series*. Springer-Verlag, 1992.

[BK89] E. Bertino and W. Kim. Indexing techniques for queries on nested objects. In *IEEE Transaction on Knowledge and Data Engineering*, 1989.

[BMG93] J. Blakeley, W. Mckenna, and G. Graefe. Experiences building the open oodb query optimizer. In *Proceedings of ACM-SIGMOD International Conference on Management of Data, 287-296.*, 1993.

[Cat93] R.G.G. Cattell. *The Object Database Standard: ODMG-93.* Morgan Kaufmann, 1993.

[CD92] S. Cluet and C. Delobel. A general framework for the optimization of object-oriented queries. *Proceedings of ACM-SIGMOD International Conference on Management of Data, 383-392.*, 1992.

[COA+94] A. Dogac C, Ozkan, B. Arpinar, T. Okay, and C. Evrendilek. Metu object-oriented dbms. *Advances in Object-Oriented Database Systems, Eds.Springer-Verlag.*, 1994.

[FG94] B. Finance and G. Gardarin. Rule-based query optimizer with adaptable search strategies. *Data and Knowledge Engineering*, 13(2), 1994.

[FLU94] J. Frohn, G. Lausen, and H. Uphoff. Access to objects by path expressions and rules. In *Proceedings of the 20th Very Large DataBase Conference*, 1994.

[GA93] O. Gruber and L. Amsaleg. Object grouping in eos. *Distributed Object Management*, 1993.

[GGT95] G.Gardarin, J.R. Gruser, and Z.H. Tang. Efficient processing of path expressions in object-oriented databases. *Submited to Data Engineering*, 1995.

[GM93] G. Graefe and W. McKenna. The volcano optimizer generator. In *Proceedings of 9th International Conference on Data Engineering, 209-218.*, 1993.

[HCF+89] L. M. Haas, W. F. Cody, J. C. Freytag, G. Lapis, B. G. Lindsay, G. M. Lohman, K. Ono, and H. Pirahesh. Extensible query processing in startburst. In *Proceedings of ACM SIGMOD International Conference on Management of Data, 377-388.*, 1989.

[IK90] Y. Ioannidis and Y. Cha Kang. Radomized algorithms for optimizing large join queries. In *Proceedings of ACM-SIGMOD International Conference on Management of Data, 312-321.*, 1990.

[JWKL90] B. Jenq, D. Woelk, W. Kim, and W.-L. Lee. Query processing in distributed orion. *In Advances in Database Technology-EDBT'90. Springer-Verlag,169-187.*, 1990.

[Knu68] D. E. Knuth. *The Art of Computer Programming*, volume 1. Addition-Wesley, 1968.

[LV91] R. Lanzelotte and P. Valduriez. Extending the search strategy in a query optimizer. In *Proceedings of ACM-SIGMOD International Conference on Management of Data, 287-296.*, 1991.

[MCS88] M. V. Mannino, P. Chu, and T. Sager. Statistical profile estimation in database systems. *ACM Computing Surveys*, 20(3):191, September 1988.

[MDZ93] G. Michell, U. Dayal, and S. Zdonik. Control of an extensible query optimizer: A planning-based approach. *Proceedings of 19th International Conference on Very Large Databases, 517-528.*, 1993.

[Mel93] J. Melton, editor. *ISO/ANSI Working Draft Database SQL (SQL3).* X3H2-93-091 ISO DBL YOK-003, 1993.

[Obj94] ObjectStore, editor. *ObjectStore User Guide Release 3.0.* 1994.

[PSC84] G. Piatetsky-Shapiro and C. Connell. Accurate estimation of the number of tuples satisfying a condition. In *Proc. ACM SIGMOD Conf.*, page 256, Boston, MA, June 1984.

[SAC+79] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *Proc. ACM SIGMOD Conf.*, page 23, Boston, MA, May-June 1979. Reprinted in M. Stonebraker, Readings in Database Sys., Morgan Kaufmann, San Mateo, CA, 1988.

[Swa89] A. Swami. A validated cost model for main memory databases. *Perfomance Evaluation Review*, May 1989.

[Yao77] S. B. Yao. Approximating the number of accesses in database organizations. *Comm. of the ACM*, 20(4):260, April 1977.

[Zai94] M. Zait. *Optimisation de requetes relationnelles pour execution parallele.* PhD thesis, Universite Pierre et Marie Curie, PARIS VI, June 1994.