METU Object-Oriented DBMS

Asuman Dogac, Cetin Ozkan, Budak Arpinar,
Tansel Okay, Cem Evrendilek

Software Research and Development Center
Scientific and Technical Research Council of Turkiye
Middle East Technical University
06531, Ankara Turkiye

Abstract

METU Object-Oriented DBMS [1] includes the implementation of a database kernel, an object-oriented SQL-like language and a graphical user interface. Kernel functions are divided between a SQL Interpreter and a C++ compiler. Thus the interpretation of functions are avoided increasing the efficiency of the system. The compiled by C++ functions are used by the system through the Function Manager. The system is realized on Exodus Storage Manager (ESM), thus exploiting some of the kernel functions readily provided by ESM. The additional functions provided by the MOOD kernel are the optimization and interpretation of SQL statements, dynamic linking of functions, and catalog management.

An original query optimization strategy based on the object-oriented features of the language is developed. For this purpose formulas for the selectivity of a path expression, and for the cost of forward and backward path traversals are derived, and join sizes are estimated. New strategies for ordering the joins and path expressions are also developed.

A graphical user interface, namely MoodView is implemented on the MOOD kernel. MoodView provides the database programmer with tools and functionalities for every phase of OODBMS application development. Current version of MoodView allows a database user to design, browse, and modify database schema interactively. MoodView can automatically generate graphical displays for complex and multimedia database objects which can be updated through the object browser. Furthermore, a database administration tool, a full screen text-editor, a SQL based query manager, and a graphical indexing tool for the spatial data, i.e., R Trees are also implemented.

1 Introduction

In this paper we describe the METU Object-Oriented DBMS (MOOD). MOOD has a type system derived from C++, eliminating the impedance mismatch between MOOD and C++. It has a SQL-like query language (MOODSQL) and is developed on top of the Exodus Storage Manager (ESM) [ESM 92], [Car 86]. This provides MOOD the following kernel functions available through ESM :

[1] The code is available from asuman@vm.cc.metu.edu.tr

- storage management
- controlling data access and concurrency
- backup and recovery of data.

  Additionally, MOOD kernel provides the following functions :

- optimization and interpretation of SQL statements and dynamic linking of functions
- catalog management.

The main problem in designing a kernel for an object-oriented DBMS is the late binding of methods to the objects. In MOOD we propose to solve this problem by dividing the labor between an object-oriented SQL interpreter and a C++ compiler. Since database environment enforces run-time modification of schema and objects, late binding is essential.

There are two other alternatives that we considered but are not chosen. These are building a system based on a persistent programming language such as C++ or using a full C++ interpreter and extending it with DBMS functionality.

In the first alternative, all other subsystems communicate via the persistent C++ which are compiled externally. The compiled programs may be executed separately, or they may be activated by using dynamic linker (dld). The disadvantage of this alternative is that it is completely orthogonal to the nature of a database management system. A DBMS provides an on-line environment rather than a database tool box where there are calls to separately compiled programs for database operations. The advantage on the other hand, is to be able to use the full power of the chosen programming language.

The second alternative although eliminates the previous disadvantage, there is a problem of performance decrease with this alternative due to interpretation. The advantage is again to be able to use the full power of C++.

The proposed approach, on the other hand, is uniform in that interfaces access the database through SQL statements interpreted by the kernel. But the code for the member functions of the classes are not interpreted. They are separately compiled with C++ and executed by SQL interpreter through a dynamic linker(dld). The advantage of this approach is that the interpretation of the functions are avoided increasing the overall efficiency of the system.

MOOD's kernel is described in Section 2. Section 3 contains brief descriptions of MOOD data model, MOODSQL and MOOD Algebra. In Section 4, cost model parameters are given and formulas for the selectivity of path expressions are derived which forms the basis of the cost calculations for the query optimizer. Section 5 summarizes the cost analysis of basic file operations. In section 6, the costs of realizing an implicit join operation through several different techniques are presented. Section 7 contains the general description of the execution of MOODSQL queries. Section 8 describes query optimization in MOOD. MoodView, the graphical user interface is then presented in Section 9. Finally summary is presented in Section 10.

2 MOOD Kernel Implementation

The general structure of the MOOD system is shown in Figure 2.1. In MOOD, data can be defined through MOODSQL data definition language or through C++. When data is defined through MOODSQL data definition language, the definitions are stored in the CATALOG and a C++ header file is created for future compilation. To handle the case where data is defined in C++, we have modified cfront such that cfront extracts the catalog information and stores it into the CATALOG.

The basic types supported by the MOOD are Integer, Float, LongInteger, String, Char, and Boolean. The type constructors are Tuple, Set, List, and Reference. A complex type may be created by using basic types and recursive application of the type constructors. A type (or class) in the system has a unique type identifier and name. The functions *typeId(char *typeName)* and *typeName(int typeId)* return type identifier and name of a type (or class) respectively.

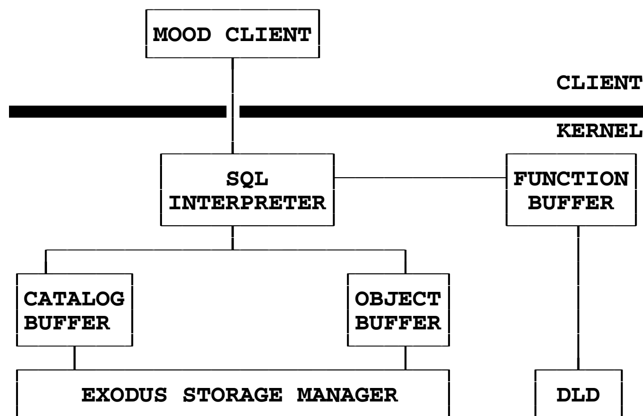


Figure 2.1. An Overview of the MOOD System

The differences between a type and a class from the implementation point of view are:

- A class has a default extent that contains the instances created.
- Values which are instances of types have copy semantic.
- Classes are organized into a class hierarchy.

The catalog contains the definition of classes, types, and member functions in a structure similar to a compiler symbol table. In order to achieve late binding at run time, it is necessary to carry compile time information to run time. This is accomplished by the use of the classes *MoodsType*, *MoodsAttribute* and

MoodsFunction. The *MoodsType* class keeps track of all the types used in the system. The *MoodsAttribute* stores the information about the attributes of these classes. The instances of the *MoodsFunction* class keeps information about the member functions. Figure 2.2 shows the structure of the catalog on ESM.

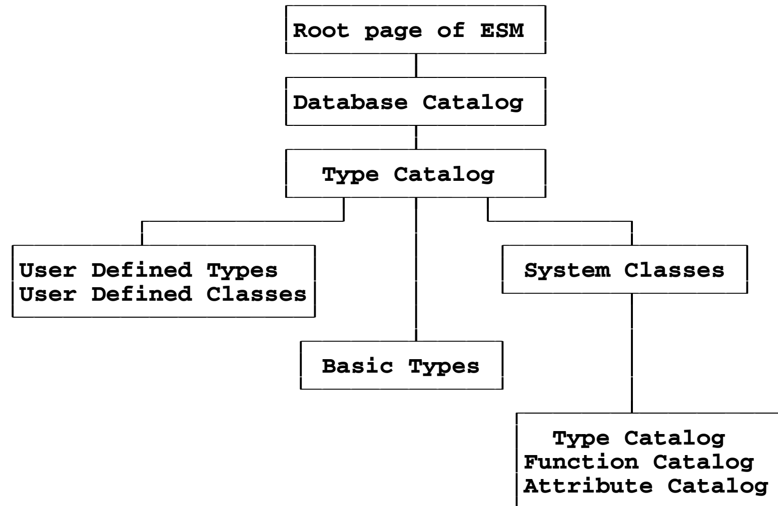


Figure 2.2. Representation of catalog in ESM

MOODSQL interpreter is responsible for :

– optimization of MOODSQL queries,
– interpretation of arithmetic and Boolean expressions,
– dynamic definition and linking of member functions.

For interpretation of arithmetic and Boolean expressions, the types of operands are necessary at run time. This information is provided by the class *OperandDataType*. As an example :

```
OperandDataType  x(INT16), y(INT32), z (DOUBLE);
x=10;
y=13;
z=(x*3+x%3)*(y/4*5) // this expression can be evaluated and result's
                    // type is casted to double since z is double.
```

The code for the interpretation of arithmetic and Boolean expressions mainly overloads addition, subtraction, multiplication, division and mode operation operators in the order (+, -,*, /, %) for arithmetic expressions. It evaluates AND, OR, NOT, and comparison operators for Boolean expressions. Type checking and conversion of results are performed at run-time.

The power of object oriented applications lies in the interpretation and late binding which necessitate the system to be an interpreter. Only by interpretation can the user requests be handled dynamically, whereas if the application is static (i.e. compiled) dynamic changes are impossible.

In our approach a Function Manager responsible for adding, updating, deleting and invoking the member functions of the classes is developed. The basic concept is to store the C++ source after some processing into the class hierarchy. In order to compile newly defined functions, MOOD keeps track of the textual definition of classes in the hierarchy. Starting from the root class in a directory hierarchy, every class has its own directory containing its textual definition and function object files and a shared object. A class named FUNCTION handles basic dynamic linking operations by the use of the Shared Object files. All of the basic types (i.e. Integer, Float, LongInteger, String, Char, and Boolean) are automatically replaced with MOOD type classes.

The member function declarations in the source code are extracted and inserted into the CATALOG. The information extracted is the member function signature information.

When a function is invoked through the SQL interpreter, the signature of the function is created by using class name to which the function is applied and its parameter list. This signature is used in locating the function in the CATALOG. When function signature is found in the CATALOG, Shared Object File of the Class is opened and the function is loaded into memory. At the point the function is called, parameters are passed to the function. Function is kept in memory until the scope changes in the program.

At run-time, adding a new function to the system has no effect on the server program, since it is separately pre-processed and compiled. The shared library of the class will be unavailable only during the time it takes to write the new function. We provide locking for this operation.

All system errors, including signals that terminate processes are handled by our Exception class. Thus although the functions are compiled, their error messages are handled as if they are interpreted.

With Function Manager the only cost is the preprocessing and compilation of the added functions for once. It is clear that during this process the server is active; there is no need to recompile the server. This is a dynamic system, where both time and memory are efficiently utilized because the interpretation time is saved and the code is loaded into memory when it is requested.

3 MOODSQL

As part of MOOD, a SQL-like object-oriented query language, MOODSQL, together with a query optimizer has been designed and implemented.

Several SQL like query languages for OODBMSs have been proposed such as CQL++ [Dar 92], O2Query [Deu 91], EXCESS [Car 88], and the query language of ORION [Kim 90].

3.1 An Overview of MOOD Data Model and MOODSQL

A detailed description of the MOOD data model and MOODSQL is given in [Ozk 93]. In this section the general structure of the data model and the query language are presented briefly.

In the MOOD data model the basic data types are Integer, Float, LongInteger, String, Char, and Boolean. Any complex data type is defined using these types and by the recursive application of the Tuple, Set, List and Reference type constructors. The data model also supports multiple inheritance and strongly typed methods. MOODSQL is designed to support ad-hoc queries in MOOD. General syntax of the query language is as follows:

```
SELECT projection-list
FROM   class-name r1,
       class-name r2,
       ...
       class-name rn

[ GROUP BY attribute-list [ HAVING predicate ] ]
[ WHERE search-expression ]
[ ORDER BY attribute-list ]
```

As an example, consider the database schema given below, which also illustrates the data definition language of MOODSQL. The classes have extensions.

```
CREATE CLASS Vehicle
    TUPLE (
        id Integer,
        weight Integer,
        drivetrain REFERENCE ( VehicleDriveTrain),
        manufacturer REFERENCE ( Company)
    METHODS:
        lbweight () Integer,
        weight () Integer,
    )
CREATE CLASS VehicleDriveTrain
    TUPLE (
        engine REFERENCE (VehicleEngine),
        transmission String(32)
    )
CREATE CLASS VehicleEngine
    TUPLE (
        size Integer,
        cylinders Integer
    )
CREATE CLASS Company
    TUPLE (
```

```
            name String(32),
            location String(32),
            president REFERENCE (Employee)
        )
    CREATE CLASS Employee
        TUPLE (
            ssno Integer,
            name String(32),
            age Integer
        )
    CREATE CLASS Automobile
        INHERITS FROM Vehicle
    CREATE CLASS JapaneseAuto
        INHERITS FROM Automobile
    int Vehicle::lbweight()
    { return weight*2.2075; }
    int Vehicle::weight()
    { return weight; }
```

MOOD System handles the methods only by keeping information on their name, return type, and names and types of their parameters. The body of methods are coded in C++ and dynamically linked through Function Manager.

The following example MOODSQL query finds the automobiles which have automatic transmission, more than 4 cylinders and produced by a non-Japanese company.

```
SELECT c
FROM EVERY Automobile - JapaneseAuto c, VehicleEngine v
WHERE c.drivetrain.transmission = 'AUTOMATIC' AND
c.drivetrain.engine = v AND v.cylinders > 4
```

The minus operator in the FROM clause is used for excluding the instances of a subclass, which would otherwise be included due to an IS-A relationship.

3.2 MOOD Algebra

In this section, the definitions of the MOOD algebra operators are given. In these definitions, the parameter argi denotes a set of objects, or set of object identifiers, or an object itself. The operators are divided into three categories according to their usage:

- general operators
- collection operators
- conversion operators.

General Operators. These operators handle the naming operation in the MOOD kernel and the operations on a single object.

ObjId(o): This operator returns the object identifier of an object o.

TypeId(o) : This operator returns the type identifier of an object *o*. Note that every object in MOOD has a typeid associated with it.

Deref(oid): The dereferencing operator which is used in referring to the object with the identifier *oid*.

isA(path): The input parameter is a path expression that starts with a class name. The return value is the class name of the last attribute of the given path.

Bind(arg, aName) : The naming operator of the algebra. It gives the name *aName* to *arg*.

Collection Operators. The objects can be accessed through the following collections:

- Object identifiers stored in a set object
- Object identifiers stored in a list object
- Objects stored in extents.

Another way to access an object is to give a unique name to an object (Named Objects). Also indices (conventional indices, binary join indices, or path indices) can be used in accessing the objects. In this section we present the operators dealing with these collections.

Select(arg, P): It selects the objects from the argument *arg* satisfying the predicate P.

Table 1. The return types of the *Select* operator

arg type	Extent	Set	List	Named Obj.
return type	Extent or Set	Set	List	Named Obj.

The possible types of the argument *arg* are Extent, Set, List and a Named Object, and the returned types are given in Table 1.

IndSel(arg, index_type, P): It selects the set of object identifiers satisfying predicate P from an extent or a group of extents named as *arg* by using the index of type *index_type*. The indexing mechanisms available for simple selection are the B^+-tree indexing and hash indexing supported through the Exodus Storage Manager. The return value of this operation is a set of object identifiers.

Project(aTupleCollection, attribute_list): The project operator is similar to the relational project operator except that *aTupleCollection* may be an extent of tuple type objects, or a list or a set of object identifiers of tuple type objects in our system. In case of a list or a set, the elements are dereferenced. The result of the operator *Project* is the extent of the tuple type values projected onto *attribute_list.* Notice since MOOD allows for dynamic schema changes, it is possible to dynamically define a class for those tuple type values and to make them objects.

Join(arg_1, arg_2, join_method, P): This operator joins arg_1 and arg_2 with join predicate P using the join method identified as *join_method*. The return types

of this operator are as shown in Table 2. The *join_method* can be one of the following:

– forward traversal
– indexed join (B^+ tree index, binary join index or path index)
– backward traversal
– pointer based hash-partition join

Partition(aTupleCollection, attribute_list): This operator divides the objects in *aTupleCollection* into groups of objects with respect to *attribute_list*. Each group is composed of objects having the same value in their corresponding attribute(s). The return value is the set of groups of objects (partitions).

Sort(aTupleCollection, sort_method, attribute_list): This operator sorts the collection *aTupleCollection* with respect to *attribute_list* by using *sort_method* without duplicate elimination. The only supported *sort_method* for the time being is heap sort with merging. If *aTupleCollection* is a set or a list then the sorted set or list of the object identifiers are the result of the sort operator. In the case of an extent, the result is the sorted extent of the objects. It is clear that the set and list cases require the dereferencing of objects identifiers.

Table 2. The return types of the *Join* operator

arg_2 \ arg_1	Extent	Set	List	Named Obj.
Extent	Extent	Extent	Extent	Extent
Set	Extent	Set	Set	Set
List	Extent	Set	List	List
Named Obj.	Extent	Set	List	Object

DupElim(arg): This operator eliminates duplicates from the *arg*. The return types of this operator are given in Table 3.

Union(arg_1, arg_2) : This operator takes the union of arg_1 and arg_2 and returns the set of objects.

Intersection(arg_1, arg_2) : This operator returns the set of objects common to arg_1 and arg_2.

Difference(arg_1, arg_2) : This operator returns the set of objects in arg_1 but not in arg_2.

Table 3. The return types of *DupElim* operator

type of *arg*	**DupElim(arg)**
Set	not applicable
List	list of ordered distinct object identifiers
Extent	Extent of the distinct object according to the deep equality check

The types of the arguments for *Union, Intersection, Difference* operators are set or list and the return types are shown in Table 4. If both arguments are lists, union corresponds to array concatenation.

Table 4. The return types *Union, Intersection, Difference* operators

type of arguments	Set	List
Set	Set	Set
List	Set	List

Conversion Operators. The type conversion functions may be carried out as a result of optimization, or their usage may be forced explicitly by the user query.

asSet(arg) : This operator converts *arg* to a set. Table 5 presents the return types for this operator.

Table 5. Return types for *asSet* and *asList* operators

type of *arg*	**elements of the resulting set or list**
Extent	Object identifiers of the objects in the extent *arg*
Set	Object identifiers of the set *arg*
Set	Object identifiers of the list *arg*
Named Object	Object identifiers of the named object

asList(arg) : This operator converts *arg* to a list. Table 5 presents the return types for this operator.

asExtent(arg) : This operator converts its argument into an extent. The possible types for the *arg* are set and list and the return types are as shown in Table 6.

Table 6. Return types for the *asExtent* operator

type of *arg*	**asExtent(arg)**
Set	extent of dereferenced objects of the elements of the set
List	extent of dereferenced objects of the elements of the set

Unnest(aTupleCollection) : This operator is borrowed from the 1NF algebra. Its output is also *aTupleCollection*. As an example, consider the extent for the following tuples. e={< o1, {o2, o3}>, <o4, {o5}>} then Unnest(e) will be a new extent, e', such that e' = { <o1,o2>, <o1, o3>, <o4, o5> }

The possible types for *aTupleCollection* for the unnest operation are presented in Table 7. Note that the return type is the extent of the tuples resulting from the unnest operation for all argument types.

Table 7. Possible argument types for the *Unnest* operator

Possible argument types for *aTupleCollection*
Extent for the tuple type objects
Set(object identifiers of tuple type objects)
List(object identifiers of tuple type objects)
A tuple type object

Nest(aTupleCollection) : *Nest* operator functions as the inverse of *Unnest* operator.

Flatten(arg) : This operator is similar to the *unnest* operation in the sense that a set or a list is flattened. The *Flatten* operator converts *arg* into the set of object identifiers. Notice that the result of the Flatten operator is always a set. The following is an example to the *Flatten* operation.

Flatten({oid1, oid2}, {oid3}} = {oid1, oid2, oid3}

4 Cost Model Parameters

In this section cost model parameters are defined and calculated along with the specification of the physical parameters describing the system. These parameters are used in various selectivity calculations which form the basis of the cost function employed in query optimization. Notice that the cost model parameters are analogous to the ones given in [Kem 90].

In Table 8, the cost model parameters are presented where C is a class, A is an attribute.

The number of the total references from class C to class D through attribute A is denoted by totlinks(A,C,D) and is given by the following equation :

$$totlinks(A,C,D) = fan(A,C,D) * |C| .$$

The probability that an instance of class D is referenced by the instances of class through attribute A is

$$hitprb(A,C,D) = totref(A,C,D) \;/\; |D| .$$

In Table 9, the information kept by the system for a B^+-tree index I is shown.

Physical parameters of the disk, which are used in the cost evaluation process are as shown in Table 10 [Sal 88].

Table 8. Cost Model Parameters

Parameter	Definition
|C|	Total number of instances of C
nbpages(C)	Total number of pages in which class C is stored
size(C)	Size of an instance of class C
notnull(A,C)	The proportion of the instances in class C with attribute A being not null
fan(A,C,D)	The average number of instances of class D that are referenced by the instances of class C through attribute A
totref(A,C,D)	The total number of objects in class D which are referenced by at least one object in class C through attribute A
dist(A,C)	Number of distinct values of the atomic attribute A of class C
max(A,C)	The maximum value of the atomic attribute A of class C
min(A,C)	The minimum value of the atomic attribute A of class C

Table 9. Parameters for a B^+-tree

Parameter	Definition
v(I)	Order of the B^+ tree
level(I)	Number of levels
leaves(I)	Number of the leaves
keysize(I)	Size of the key value
unique(I)	Unique flag

4.1 Selectivity

Selectivity is a parameter used with a predicate to denote the ratio of the elements of a collection satisfying a given predicate. When optimizing the queries, selectivity of a predicate is estimated assuming that the values are uniformly distributed. The traditional uniformity and randomness assumptions about value distributions tend to overestimate costs. However more sophisticated techniques require more statistical information about the database. The question of how to maintain such information within tolerable overhead is not yet fully resolved [Jar 85].

A simple predicate in the system is a triplet of the form <P_1 , θ, oprnd>,

Table 10. Physical Parameters for hard disk

Parameter	Definition
B	block size
btt	block transfer time
ebt	effective block transfer time
r	average rotational latency
s	average seek time

where P_1 is a path expression, θ is a comparison operator (=, <>, >=, <=, >, <), and oprnd is either a constant or another path expression.

Selectivity for Atomic Attributes The well-known selectivity calculations assuming the uniform distribution of the atomic values described in [Ozk 90] will be used throughout the derivations presented in this paper.

The selectivity of the expression "s.A = constant", denoted f_s, where s is a bind variable of class C, and A is an atomic attribute, is given by the following formula:

$$f_s(s.A) = 1 \; / \; dist(A,C)$$

The selectivity of the expression " s.A > constant " is

$$f_s(s.A) = (\; max(A,C) \text{ - } constant \;) \; / \; (\; max(A,C) \text{ - } min(A,C) \;)$$

The selectivity of the expression "s.A BETWEEN $cons_1$ and $cons_2$" is

$$f_s(s.A) = (\; cons_2 \text{ - } cons_1 \;) \; / \; (\; max(A,C) \text{ - } min(A,C) \;).$$

Selectivity of Path Expressions Assume that there exists a path expression that contains m attributes, A_1 through A_m, A_1 through A_{m-1} being constructed using set and reference constructors, and A_m is an atomic attribute, A_i being an attribute of class C_i. Then, for the single path expression predicate "$p.A_1.A_2...A_m \; \theta$ c", where θ is a comparison operator and c is a constant, the selectivity, $f_s(p.A_1.A_2...A_m, \theta)$ is to be calculated. For this calculation we define the shorthand notation for some of the previously mentioned parameters as follows:

$$\begin{aligned} fan_i &= fan(A_i, C_i, C_{i+1}) \\ totref_i &= totref(A_i, C_i, C_{i+1}) \\ totlinks_i &= totlinks(A_i, C_i, C_{i+1}) \quad where \;\; 1 \leq i \leq m\text{-}1. \end{aligned}$$

The calculation of the selectivity of "$A_m \; \theta$ c", $f_s(A_m)$, is clear from the previous section. Therefore the expected number of instances of C_m, denoted by k_m, satisfying "$A_m \; \theta$ c" is:

$$k_m = | \; C_m \; | \; * f_s(A_m)$$

In forward traversal, assuming that we start with k objects of class C_1 and traverse the path $p.A_1.A_2...A_i$ in forward direction , the expected number of objects of class C_{i+1} , denoted by fref, is given by the following formula:

$$fref(p.A_1...A_i, k) = \begin{cases} k & ,i = 0 \\ c(totlinks_i, totref_i, fref(p.A_1...A_{i-1}, k) * fan_i) & ,i > 0 \end{cases}$$

where, c(n,m,r) is defined to be an approximation to the number of different colors, when r objects are chosen out of n objects uniformly distributed over m colors given as follows [Cer 85].

$$c(n,m,r) = \begin{cases} r & , r < \frac{m}{2} \\ \frac{(r+m)}{3} & , \frac{m}{2} \leq r < 2m \\ m & , r \geq 2m \end{cases}$$

Note that better approximations to this problem are given in [Yao 77], [Car 75]. However it has been validated that *c(n,m,r)* well serves our purposes. Starting with one instance of class C_1, we denote the number of objects of class C_m obtained at the end of forward path traversal by fref(p.A_1...A_{m-1},1). On the other hand, k_m objects have been selected through the predicate A_m θ c. Then the selectivity of a path expression which is assumed to be the probability of at least one object being in common in two sets with cardinalities fref(p.A_1.A_2...A_{m-1},1) and k_m*hitprb(A_{m-1},C_{m-1},C_m) respectively, is given by

$$f_s(\ p.A_1.A_2...A_m\ ) = o(totref_{m-1}\ ,\ fref(p.A_1.A_2...A_{m-1}\ ,1), k_m * hitprb(A_{m-1}, C_{m-1}, C_m))$$

where o(t,x,y) is the probability that there exists at least one common object in two sets selected with replacement out of t distinct objects and is defined as

$$o(t,x,y) = 1 - C(t-x,y) / C(t,y)$$

where C stands for combination, and x and y are the cardinalities of the two sets respectively. The term *C (t-x,y) / C(t,y)* is the probability that the sets with cardinalities x and y never intersect.

5 Cost Analysis of Basic File Operations

In this section the costs of sequential, random and indexed accesses are given. The cost of sequential access to *b* pages is denoted by *SEQCOST(b)* and calculated as

$$SEQCOST(b) = s + r + b * ebt\ .$$

The assumption that pages of a file are stored consecutively on disk may not be true in all systems. For example in ESM, a file is stored as a B+ tree and therefore the sequential access cost of a file is equal to its random access cost.

The cost of random access to *b* pages, denoted by *RNDCOST(b)* is

$$RNDCOST(b) = b * (\ s + r + btt\ ).$$

The cost of accessing object identifiers for *k* random keys from a secondary index *I*, referred to as *INDCOST(k)*, is

$$INDCOST(k) = \left(\sum_{i=1}^{level(I)} \lceil c(n_i, m_i, r_i) \rceil\right) * RNDCOST(1)$$

where n_i = leaves(I) / $(2v(I)*\ln 2)^{i-2}$, m_i = leaves(I) / $(2v(I)*\ln 2)^{i-1}$, and

$$r_i = \begin{cases} k & , i = 1 \\ c(n_{i-1}, m_{i-1}, r_{i-1}) & , otherwise. \end{cases}$$

Finally, the cost of a range query using a B^+-tree index I, given by *RNGX-COST(fract)*, is obtained by the equation

$$RNGXCOST(fract) = fract * leaves(I) * (s + r + btt)$$

where *fract* is the proportion of the specified range to the whole domain.

6 Cost of Implicit Join Operation

Throughout the following cost analysis, it is assumed that k_c objects of Class C are to be joined (implicitly) through the attribute A of the class C with k_d objects of Class D, which can be explicitly shown as C.A=D.self. Note that if there are no previous selections $k_c = |C|$ and $k_d = |D|$.

6.1 Cost of Forward Traversal

When there are k_c*fan(A,C,D) number of references from class C to D induced by a group of k_c objects from C, the expected number of pages of class D to be retrieved to realize the join operation is given by the following formula:

$$ftc = RNDCOST(nbpg_c) + RNDCOST(k_c * fan(A,C,D))$$

where $nbpg_c = nbpages(C) * (1 - (1 - 1/nbpages(C))^{k_c})$.

This is worst case formula where there are no page hits in the buffer allocated for the objects of Class D.

6.2 Cost of Backward Traversals

In order to backward traverse k_d objects of the class D into k_c objects of class C, to perform the join C.A=D.self, it is necessary to make a sequential scan over the extent of class C. Therefore the cost is given by the following formula:

$$btc = SEQCOST(nbpages(c)) + k_c * fan(A,C,D) * k_d * CPUCOST + \begin{cases} 0 & \text{if D is accessed previously} \\ SEQCOST(nbpages(D)) & else \end{cases}$$

6.3 Cost of Using Binary Join Indices

The cost of using a binary join index for k objects of either class C or D is

$$bjc = INDCOST(k)$$

6.4 Cost of Using Pointer-based Hash Partition Join

The cost of Hash-Partition Join in relational databases for two relations is given as [Sal 88]

$$3*(b+b')*ebt+[1+(b'/b)]*nsg^2*(r+s)$$

where *b* and *b'* are the number of blocks in outer and inner relations respectively, and *nsg* denotes the number of segments of the smaller relation.

In case of pointer-based Hash-Partition Join, the referencing class, i.e., class C is hashed on the pointer field A and partitions are created. Then for each object of class C, the pointer, C.A, is chased to retrieve the object from class D. So the cost of joining k_c objects of class C with the objects of class D by using pointer-based hybrid hash join can be given as follows:

$$hhc = 3* k_c/\,|C|*SEQCOST(nbpages(C) + RNDCOST(nbpg)\ )$$

where $nbpg = nbpages(D) * (\ 1 - (\ 1 - 1/nbpages(D))^{\alpha})$ and α=c(|C|*fan(A,C,D), totref(A,C,D), k_c*fan(A,C,D)).

Note that this join technique can only be applied when constructor of attribute A is Reference.

7 General Description of the Execution of MOODSQL Queries

The MOODSQL queries are executed in a predefined order of MOODSQL clauses. The sequence of execution of these clauses is shown in Figure 7.1. It should be noted that, this order is, in fact, implied by the nature of the language.

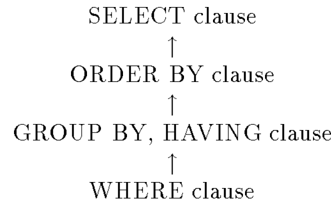


Figure 7.1. The sequence of execution of a MOODSQL query

Also within a WHERE clause, there is a predefined order of algebraic operators as shown in Figure 7.2. In our implementation, this order is enforced by the nature of our object oriented data model together with the well-accepted query optimization principles in relational DBMSs.

Usually, after generating the parse tree of a query, it is modified according to the algebraic transformations to obtain the final optimized tree [Ull 88]. In

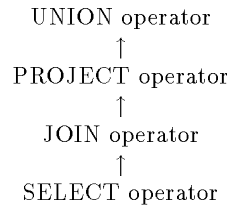


Figure 7.2. Order of execution of algebraic operators in a WHERE clause

our implementation, the final tree structure is created in a single pass with the help of the tables constructed during query parsing, along with the cost calculations performed for optimization. The query processor augmented with the query optimizer works as follows :

- The query is parsed.
- The expressions are simplified.
- The predicates in the WHERE and HAVING clauses in the query are transformed into disjunctive normal form. Such an expression takes the form

(p_{11} AND p_{12} AND ... AND p_{1m}) OR (p_{21} AND p_{22} AND ... AND p_{2r}) OR ...where each p_{ij} is a predicate and (p_{i1} AND p_{i2} AND ...) is called an AND-term. Thus, the UNION operation is performed after evaluating the predicates for the AND-terms.

We classify the selection predicates into three types.

Immediate Selection : Selection depends on an atomic attribute or a parameterless method. These type of predicates have the following form "s.A θ c", where s, A, θ, and c are a range variable, an atomic attribute or a parameterless method, a comparison operator, and a constant respectively. The information related to these kinds of predicates are kept in a dictionary, called ImmSelInfo. The structure of the ImmSelInfo dictionary is given in Table 11.

Table 11. Structure of the Dictionary ImmSelInfo

Range Variable	Predicate	Selectivity	Indexed Access Cost	Sequential Access Cost	Access Type

Path Selection: Selection depends on a path expression. General form for this type is "s.A_1...A_m θ c ", where s.A_1...A_m, θ, and c are a path expression, a comparison operator, and a constant respectively. Notice that a path expression implies an implicit join. The structure of PathSelInfo dictionary is given in Table 12.

Other Selections: Other predicates which are not classified as one of the above types. The examples for such predicates include methods and complex

Table 12. Structure of the Dictionary PathSelInfo

Range Variable	Predicate	Selectivity	Forward Traversal Cost

predicates. The main problem for this type is that it is not so easy to calculate the selectivity, and hence, the cost. The related information is stored in the OtherSelInfo dictionary. The data structure for this dictionary is also the same as that of ImmSelInfo.

- The optimal execution order of immediate and path selections are decided as explained in Section 8.
- To decide on a near optimal execution plan among feasible alternatives, a new heuristic is developed for ordering joins as explained in Section 8. After executing the joins projections are performed.
- Finally, all the subaccess plans generated are combined using the UNION operation.

8 Query Optimization in MOOD

Currently, query optimization for object-oriented database management systems is a challenging research area. The goal of the query optimization is to find an execution plan for a specific query in order to minimize a cost function. The steps involved in this process can be considered at two levels, the logical query optimization (query rewriting) that uses semantic properties of the language in order to find expressions equivalent to the one given by the user and the physical query optimization, that is based on a cost model to choose the best algorithm for evaluating the query [Clu 92].

In the following, optimization strategies of MOODSQL are presented.

8.1 Ordering of Atomic Selections

For the immediate selection predicates in the ImmSelInfo dictionary, the selectivity of the predicates and the cost of sequential scan for each range variable are calculated. For predicates involving an indexed attribute, indexed access costs are calculated and for each range variable in an AND term these index costs are sorted in the ascending order. Note that for common predicates in AND terms, the cost is calculated only once. Let $\mathrm{cost_i}$ denote the indexed access cost of $\mathrm{i^{th}}$ item in the sorted order which is calculated as follows:

$$cost_i = \begin{cases} INDCOST(1) & , \theta \text{ is "="} \\ RNGXCOST(f_s(P_i)) & , otherwise \end{cases}$$

where $\mathrm{f_s}$ $(\mathrm{P_i})$ is as given in Section 4.1. Then the number of indices to be used is the maximum value k satisfying the inequality

$$\sum_{i=1}^{k} cost_i + RNDCOST(|\mathrm{C}| * \prod_{i=1}^{k} f_s(P_i)) < SEQCOST(nbpages(C))$$

where C is the class to which the range variable is bound.

The remaining predicates for each range variable in an AND term are sorted in increasing order of their estimated selectivities and applied in this order. The heuristic used here is analogous to short circuiting used in compilers for Boolean expression evaluation: evaluating the predicates from the least selective to the most so that minimum number of predicates are evaluated for each object.

8.2 Determining the Optimum Execution Order of Path Expressions

Given m path expressions in an AND-term :

$p.a_{11}.a_{12}...a_{1n_1}$

$p.a_{21}.a_{22}...a_{2n_2}$

.

.

$p.a_{m1}.a_{m2}...a_{mn_m}$

the problem of finding the least costly execution order of these path expressions can be stated as the following minimization problem:

Find a permutation of the integers 1 through m stored in i[1] through i[m] which minimizes

$$f = F_{i[1]} + s_{i[1]}*F_{i[2]} + s_{i[1]}*s_{i[2]}*F_{i[3]} + ... + s_{i[1]}*s_{i[2]}* ... s_{i[m-1]}*F_{i[m]}$$

where F_j and s_j, j i[1] through i[m], are the cost of traversing and the selectivity of the jth path expression respectively. In other words, we are trying to minimize the objective function f, denoting the total cost of executing m path expressions in the order induced by the array i.

Assume π denotes a permutation of the integers 1 through m such that path expression indices are sorted in ascending order of F_i / (1 - s_i) values, such that $1 \leq i \leq m$. This π minimizes the objective function f.

The formal treatment of this problem is provided in the Appendix.

Algorithm 8.1. The Evaluation Order of Path Expressions

- Calculate the forward traversal cost F_i for each path expression.
- Calculate the forward traversal selectivity (s_i) for each path expression.
- Order the path expressions by sorting them according to F_i / (1-s_i) values.

Example 8.1

Let us assume that the statistics given in Tables 13,14 and 15 have been collected for an example database.

Consider the example query:

Select v

From Vehicle v

where v.company.name = 'BMW' and v.drivetrain.engine.cylinders = 2

The PathSelInfo dictionary for the example query is given in Table 16.

The order of the path expressions is P2 followed by P1. After applying algorithm 8.2 given in section 8.3 to the path expression P2, the following subaccess plan is generated.

Table 13. Statistics on the example database

Class	|C|	nbpages(C)	size(C)
Vehicle	20000	2000	400
VehicleDriveTrain	10000	750	300
VehicleEngine	10000	5000	2000
Company	200000	2500	500

Table 14. Statistics on the example database

Class	Attribute	dist	max	min
VehicleEngine	cylinders	16	32	2
Company	name	200000	-	-

```
T1 : JOIN(
     BIND(Vehicle, v),
     SELECT(BIND(Company, c), c.name = 'BMW'), HASH_PARTITION,
     v.company = c.self )
```

Then applying algorithm 8.1 by taking into account the effect of the path expression P1, the following access plan is generated.

```
JOIN(
       JOIN( T1, BIND(VehicleDriveTrain,d),
                   FORWARD_TRAVERSAL, v.drivetrain = d.self ),
       SELECT(BIND(VehicleEngine, e), e.cylinder=2),
FORWARD_TRAVERSAL, d.engine = e.self )
```

8.3 Join Optimization

Join optimization, i.e., deciding on the execution order of join operations is one of the most important decisions that is made by the optimizer. Join optimization does not become a simpler problem due to the precomputed joins (stored references) and path indices; instead, it becomes a more complex problem because the number of join strategies grows with the number of alternative access paths [Bla 93]. In this section we propose an algorithm for ordering implicit joins in a path expression. In realizing the implicit joins one of the following join strategies is used.

Table 15. Statistics on the example database

Class	Attribute	fan	totref	totlinks	hitprb
Vehicle	drivetrain	1	10000	20000	1
Vehicle	manufacturer	1	20000	20000	0.1
VehicleDriveTrain	engine	1	10000	10000	1

Table 16. PathSelInfo dictionary contents for Example 8.1

Range Variable	Predicate	f_s	Forward Traversal Cost	cost/(1-f_s)
v	P1:v.drivetrain. engine.cylinders=2	6.25e-2	771.825	823.280
v	P2:v.company. name='BMW'	5.00e-5	520.825	520.825

- Forward traversal
- Backward traversal
- Index-based join (Binary Join Indices)
- Pointer-based hash partition join

Let us assume that there is a path expression $p.a_1.a_2...a_n$ where p is bound to C_0 and a_i references to the instances of the class C_i ($1 \leq i \leq n-1$). jc_{ij}, and js_{ij} will denote the individual cost and selectivity of the temporary collection C_{ij} obtained from joining class C_i and class C_j respectively. is the set of all classes which are candidates for join in each iteration of the algorithm.

We use a greedy heuristic in solving this problem such that at each iteration the join pair with lowest cost and highest selectivity are favored at the same time. The function f(jc, js)=jc/(1-js) where jc is the cost of the join and js is the selectivity of the join, satisfies the required selection criterion.

Algorithm 8.2. Implicit Join Ordering

In this algorithm jc is the minimum cost join technique among the four join algorithms given above.

1. Initialize the list Δ with the classes in the path expression
$\Delta = \{ C_0, C_1, ..., C_{n-1}\}$
2. For i=0 to n-2 compute $jc_{i,i+1}$ and $js_{i,i+1}$.
3. Sort the items with respect to jc / 1-js values in ascending order.
4. Select the minimum pair <C_i, C_{i+1}> in the sort order and name it C_k.
5. Delete i^{th} and $i+1^{st}$ items.
6. Compute $jc_{i-1,k}$, $js_{i-1,k}$, $jc_{k,i+2}$, and $js_{k,i+2}$.
7. Insert <C_{i-1}, C_k> and <C_k, C_{i+2}> to the sorted list.
8. if the number of elements in Δ is 1 goto 9 else goto 4.
9. Generate an access plan for this join.

Example 8.2

This example is provided to clarify the implicit join ordering algorithm. Consider the query, Select v

From Vehicle v

Where v.drivetrain.engine.cylinders = 2

The initial cost and selectivity estimations for the example query are given in Table 17.

At the end of the first iteration the following subaccess plan is generated;

T1 = JOIN(BIND(VehicleDriveTrain, d),

Table 17. The initial cost and selectivity estimations for Example 8.2

CLASS	**CLASS**	**ftc**	**btc**	**bjc**	**hhc**	f_s	**cost/(1-f_s)**
Vehicle	VehicleDriveTrain	39.38	24.92	-	11.98	0.0001	11.98
VehicleDriveTrain	VehicleEngine	19.24	10.92	-	5.27	0.0001	5.27

SELECT(BIND(VehicleEngine, e), e.cylinders=2),
HASH_PARTITION, d.engine =e.self)

The final execution plan is as follows:
JOIN(BIND(Vehicle, v), T1, HASH_PARTITION, v.drivetrain = d.self).

9 MoodView, the Graphical User Interface

MoodView is the graphical front end to MOOD [Arp 93]. It allows the user to browse, edit and query the database schema and the objects. MoodView provides an environment which conforms to the requirements of object-oriented systems. It is a sophisticated, but easy to use interface. It provides visual access to the database without requiring expertise from the database user.

9.1 Design Principles

MoodView is based on the graphical direct manipulation paradigm and is implemented in C++ using X Windows/Motif toolkit.

Conventional interfaces such as C++ and SQL are also integrated into the GUI (Graphical User Interface). Class hierarchy of C++ code can be displayed visually and C++ code can be generated from the visual definitions. MoodView provides a SQL based query formulation tool, that the user can prepare her queries and receive results in a graphical environment.

All the schema information maintained by MoodView is stored in the database catalog through the system defined classes. MoodView actions are performed through execution of the methods by the MOOD Kernel on these data.

9.2 Environment

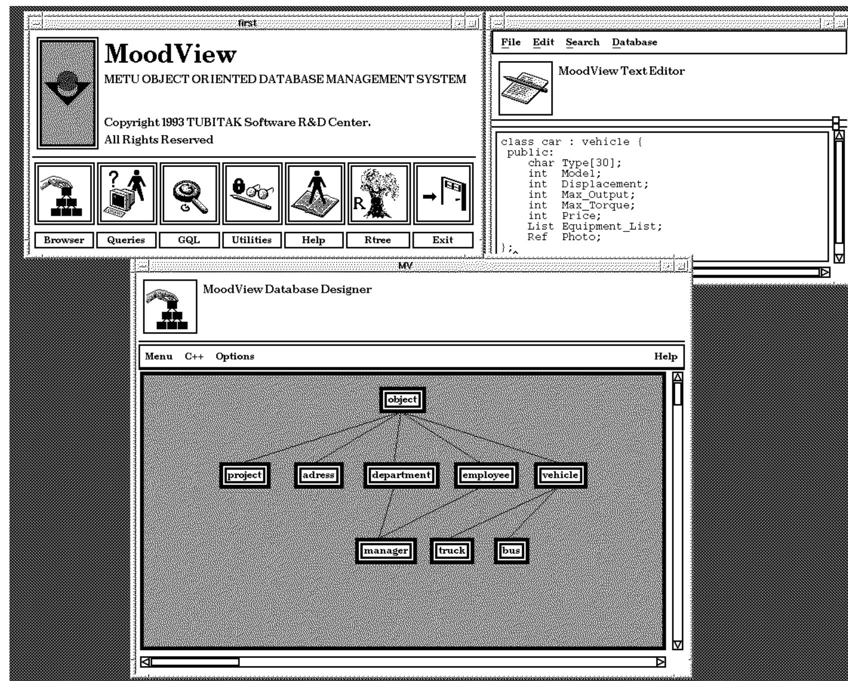
Upon entering the programming environment, an initial window that contains the icons for each of the MoodView tools is displayed as shown in Figure 9.1(a).

Schema Browser. A database schema in MOOD contains class types, their methods and relationships between those classes. Their inheritance relationships is represented as a DAG (Directed Acyclic Graph) and MoodView uses a DAG placement algorithm that minimizes crossovers and makes drawings for graph nodes as shown in Figure 9.1(c). It allows the user to design, browse, and modify database schema interactively.

Data Definition in C++. MoodView can display a class hierarchy defined in C++ (Figure 9.1(b)). Cfront of C++ translator is modified such that,

(a)

(b)



(c)

Figure 9.1.(a) Initial MoodView Window,**(b)**Data Definition in C++, **(c)** Class Hierarchy Browser

when data is defined through C++, cfront extracts the schema information and MoodView can display class hierarchy graphically using the output of Cfront. MoodView also can convert graphically designed class hierarchy graph into C++ code.

Attributes. Class attributes can be updated through a tool for designing object oriented data types. One can add, drop attributes, change the name or the type of an attribute by using this tool as shown in Figure 9.2(c).

Methods. MoodView allows creation and deletion of methods and update of existing method bodies and parameters as shown in Figure 9.2(a).

9.3 Object Browsing

MoodView allows complex operations against a set of objects. These include creation, deletion, update and automatic display of complex and multimedia objects, and the invocation of methods. Projection, selection and complex query specification can be done on the objects through the SQL based query manager.

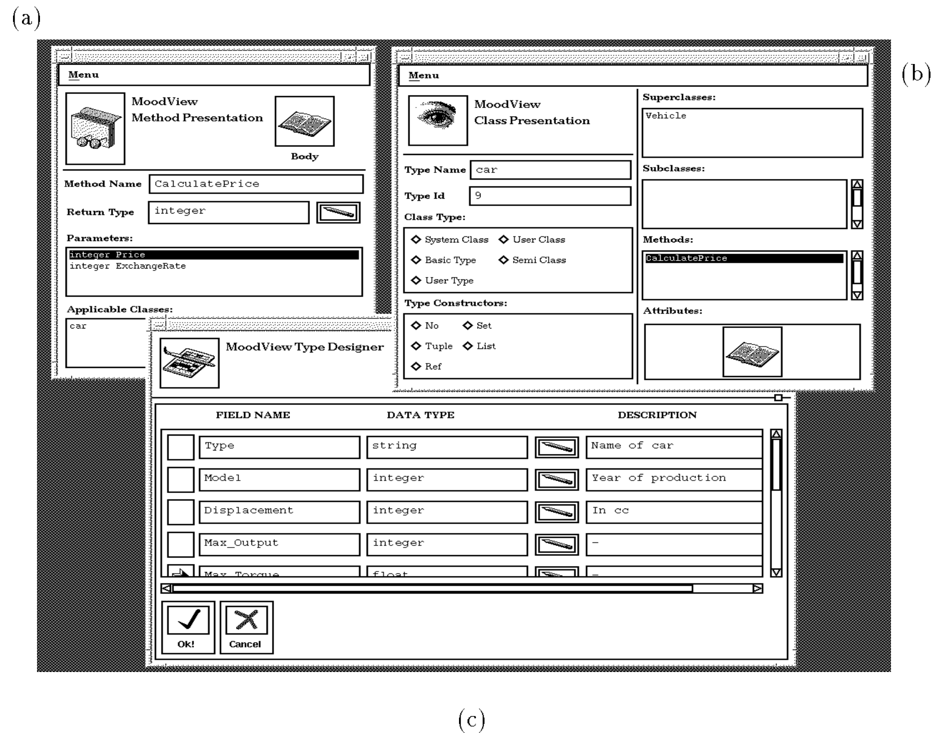


Figure 9.2.(a) Method Presentation,**(b)**Class Presentation, **(c)** Class Designer

Generic Object Presentations. MOOD objects constitute graphs connecting atoms and constructors. MoodView has a generic display algorithm for displaying these object graphs and walking through the referenced objects. Multimedia data such as images in different formats is defined through the system classes.

Updates to Object Presentations. Atomic types such as string or integer can be the widgets representing complex types. Copy, paste operations are also allowed for both atomic and complex types. Dynamic type checking is performed by MoodView to ensure the correctness of updates.

Interactive Method Activation. Methods are attached to object presentations and can be activated interactively.

Query Formulation. Query manager provides a query editor with facilities for accessing previous queries in a session. Through queries, objects with specific characteristics (selection) or selected portions of the objects (projection) can be displayed graphically.

(a) (b)

Figure 9.3.(a) Generic Presentation For a Car Object,**(b)**Generic Presentation For the Car Objects

9.4 Implementation

MOOD Kernel interprets provides all the functions needed by MoodView to manage schema and instance levels.

Use of the Catalog. The MOOD catalog contains all of the information required to manage schema through MoodView such as the definition of classes, types, and member functions in a structure similar to a compiler symbol table. The design of MOOD Catalog makes MoodView easily extendible, therefore it can be used in a straightforward manner for new types and objects added to the MOOD. For example, MoodView uses this persistent type catalog to determine how an object of certain type is to be displayed.

SQL Interface Between The Kernel and MoodView. A standard communication protocol is choosen between the database kernel and the GUI. All the database operations performed by the user through MoodView are converted to SQL statements and the interpretation of SQL statements is performed by the Kernel. For example, to create a new instance of employee class, MoodView produces the following SQL statement:

new Employee < "Budak Arpinar", "Computer Engineer", 1969>

Object Presentations. A cursor like mechanism which exists commonly in RDBMSs is designed for displaying objects. It is the Kernel's responsibility to identify type and value of an object in the system at run-time using the MOOD Catalog. The kernel gets the stored representation of the object from the database and returns a pointer to a buffer area each element of which specifies

a name, a type and a value of the object's attributes. MoodView synthesizes this information and combines widgets to display an object on the screen. It is also possible to sequence back and forth through the returned objects using the cursor functions provided by the kernel.

10 Summary

We have discussed the design and implementation of an object-oriented DBMS, namely METU Object-Oriented DBMS.

The system includes the following components:

- A database kernel which is responsible from catalog management and dynamic linking of functions.
- An object algebra
- An object-oriented query language, MOODSQL, and its optimizer
- A graphical user interface.

Acknowledgements

The authors wish to gratefully acknowledge the MOOD project implementation team: Mehmet Altinel, Ilker Altintas, Ilker Durusoy, Tolga Gesli, Ismail Tore and Yuksel Saygin.

References

[Arp 93] Arpinar, I. B., Dogac, A., Evrendilek, C., "MoodView: An Advanced Graphical User Interface for OODBMSs", SIGMOD Record, Vol. 22, No. 4, December 1993.

[Bla 93] Blakely, J., Mc Kenna, W. J., Graefe, G., "Experiences Building the Open OODB Query Optimizer" in *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, 1993.

[Car 75] Cardenas A.F., "Analysis and Performance of Inverted Data Base Structures", *Comm. of ACM*, Vol. 18, No. 5, May 1975.

[Car 86] Carey, M., DeWitt, D.. Richardson, J., Shekita, E., "Object and File Management in EXODUS Extensible Database System", in *Proc. of the 12th Intl. Conf. on VLDB*, 1986.

[Car 88] Carey M.J., DeWitt D.J., Vandenberg S.L., "A Data Model and Query Language for EXODUS", *Proc. of the ACM SIGMOD Conf.*, 1988.

[Cer 85] Ceri, S., Pelagatti, G., *Distributed Database Systems*, McGraw Hill,

1985.

[Clu 92] Cluet S., Delobel C., "A General Framework for the Optimization of Object-Oriented Queries", in *Proc. of the ACM SIGMOD Conf. on Management of Data*, 1992.

[Dar 92] Dar S., Gehani N.H., Jagadish H.V., "CQL++: A SQL for the Ode Object-Oriented DBMS", in *Proc. of Extending Database Technology*, 1992.

[Deu 91] Deux, O., et al., "The O2 System", *Comm. of the ACM*, Vol. 34, No.10, 1991.

[ESM 92] Using the Exodus Storage Manager V2.1.1, June 1992.

[Jar 85] Jarke M., Koch J., Schmidt J. W., "Introduction to Query Processing", in *Query Processing in Database Systems*, Springer-Verlag, 1985, pp 3-28.

[Kem 90] Kemper A., Moerkotte G., " Access Support in Object Bases", in *Proc. of the ACM SIGMOD Conf.*, 1990.

[Kim 90] Kim W., *Introduction to Object-Oriented Databases*, The MIT Press, 1992.

[Oka 93] Okay, T., Ozkan, C., Dogac, A., "Design and Implementation of a Database Kernel", Tech. Rep. 15, TUBITAK Software Research and Development Center, March 1993.

[Ozk 93] Ozkan, C., Dogac, A., Evrendilek, C., Gesli, T., "Efficient Ordering of Path Traversals in Object-Oriented Query Optimization", In Proc. of the Intl. Symp. on Computer and Information Sciences, Istanbul, November 1993.

[Ozk 90] Ozkarahan E., *Database Management Concepts, Design and Practice*, Prentice-Hall, 1990.

[Sal 88] Salzberg B., *File Structures, an Analytic Approach*, Prentice-Hall, 1988.

[Ull 88] Ullman J. D., *Principles of Database and Knowledge-Base Systems*, Vol. 2, Computer Science Press, 1988.

[Yao 77] Yao S.B., "Approximating Block Access in Database Organizations", *Comm. of ACM*, Vol. 20, No. 4, April 1977.

APPENDIX

Given m path expressions in an AND-term, the problem of finding the least

costly execution order of these path expressions may be stated as the following minimization problem.

Find a permutation of the integers 1 through m stored in i[1] through i[m] which minimizes
$f = F_{i]1[} + s_{i]1[} {}^{*}F_{i]2[} + s_{i]1[} {}^{*}s_{i]2[} {}^{*}F_{i]3[} + \ldots + s_{i]1[} {}^{*}s_{i]2[} {}^{*} \ldots s_{i]m-1[} {}^{*}F_{i]m[}$
where F_j and s_j, j i[1] through i[m], are the cost of traversing and the selectivity of the jth path expression respectively.
Lemma : Assume π denotes a permutation of the integers 1 through m such that path expression indices are sorted in ascending order of F_i / (1 - s_i) values, such that $1 \leq i \leq m$. This π minimizes the objective function f.
Sketch of Proof : By induction on the number of path expressions.
It is true for 2 path expressions.
In this case $f=F_1+ s_1 F_2$ or $f= F_2 + s_2 F_1$.
if $F_1+ s_1 F_2 < F_2 + s_2 F_1$ then by simple manipulation,
$F_1 / (1 - s_1) < F_2 / (1 - s_2)$ is found.

Assume it is true for m path expressions and try to show that it is also true for m+1 path expressions. Let us assume that $F_i/(1-s_i) < F_{i+1} (1-s_{i+1})$ for $1 \leq i \leq m-1$, and assume also that $F_j/(1-s_j) < F_{m+1}/(1-s_{m+1}) < F_{j+1}/(1-s_{j+1})$ for some j where $1<j<m$.
We claim that
$f_1=F_1+s_1F_2+\ldots+s_1s_2\ldots s_{j-1}F_j+s_1s_2\ldots s_{j-1}s_jF_{m+1}+s_1s_2\ldots s_{j-1}s_js_{m+1}F_{j+1}+\ldots+s_1$ minimum. Assume on the contrary that,
$f_2=F_1+s_1F_2+\ldots+s_1s_2\ldots s_{k-1}F_k+s_1s_2\ldots s_{k-1}s_kF_{m+1}+s_1s_2\ldots s_{k-1}s_ks_{m+1}F_{k+1}+\ldots+s_1$ minimum with the assumption that $k<j$ without loss of generality.

First observe that by the induction hypothesis, it can be shown that with the addition of the $m+1^{st}$ path expression, the relative order of the previous path expression indices do not change.

Therefore, if we parenthesize f_2 by $s_1s_2\ldots s_{k-1}s_k$ starting from the $k+1^{st}$ term, we observe that the induction hypothesis stating that aforementioned sort order minimizes the objective function for $m+1-k \leq m$ path expressions , is violated.

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