Implementation Aspects of an Object-Oriented DBMS

Asuman Dogac Mehmet Altinel Cetin Ozkan
Ilker Durusoy
Software Research and Development Center
Scientific and Technical Research Council of Turkiye
Middle East Technical University (METU)
06531 Ankara Turkiye
email: asuman@srdc.metu.edu.tr

Extended Abstract

1 Introduction

This paper describes the design and implementation of an OODBMS, namely the METU Object-Oriented DBMS (MOOD). MOOD [Dog 94b] is developed on the Exodus Storage Manager (ESM) [ESM 92] and therefore some of the kernel functions like storage management, concurrency control, backup and recovery of data were readily available through ESM. In addition ESM has a client-server architecture and each MOOD process is a client application in ESM. The kernel functions provided by MOOD are the optimization and interpretation of SQL statements, dynamic linking of functions, and catalog management. SQL statements are interpreted whereas functions (which have been previously compiled with C++) within SQL statements are dynamically linked and executed. A query optimizer is implemented by using the Volcano Query Optimizer Generator. A graphical user interface, namely MoodView [Arp 93a, Arp 93b], is developed using Motif. MoodView displays both the schema information and the query results graphically. Additionally it is possible to update the database schema and to traverse the references in query results graphically.

The system is coded in GNU C++ on Sun Sparc 2 workstations. MOOD has a SQL-like object-oriented query language, namely MOODSQL [Ozk 93b, Dog 94c]. MOOD type system is derived from C++, thus eliminating the impedance mismatch between MOOD and C++. The users can also access the MOOD Kernel from their application programs written in C++. For this purpose MOOD Kernel defines a class named UserRequest that contains a method for the execution of MOODSQL statements. The MOOD source code is available both for anonymous ftp users from ftp.cs.wisc.edu and for the WWW users from the site http://www.srdc.metu.edu.tr along with its related documents.

In MOOD, each object is given a unique Object Identifier (OID) at object creation time by the ESM which is the disk start address of the object returned by the ESM. Object encapsulation is considered in two parts, method encapsulation and attribute encapsulation. These encapsulation properties are similar to the public and private declarations of C++.

Methods can be defined in C++ by users to manipulate user defined classes and after compilation, they are dynamically linked and executed during the interpretation of SQL statements. This late binding facility is essential since database environments enforce run-time modification of schema and objects. With our approach, the interpretation of functions are avoided thus increasing the efficiency of the system. Dynamic linking primitives are implemented by the use of the shared object facility of SunOS [Sun 90]. Overloading is realized by making use of the signature concept of C++.

2 MOOD Kernel Design Considerations and an Overview

Objects are grouped in the abstraction level of a class, in other words, classes have extensions. Class extensions are implemented as ESM files. A class in the system has an unique type identifier which is inherited from a meta class named MoodsRoot. This type identifier is used in accessing the catalog to obtain the type information to be used in interpreting the ESM storage objects which are untyped arrays of bytes. The relation between classes and instances is a 1:n relation, i.e., under a class there could be any number of instances associated with it, but an instance can not be associated with more than one class. Class inheritance mechanism of the MOOD is multiple inheritance. The name resolution is handled as in standard C++. Additionally in our system in case of name conflicts, if the scope resolution operator is not

used, the first class in the inheritance order having that attribute is assumed as default.

Aggregate definitions are handled in the MOOD system by introducing type constructors (Set, List, Ref and Tuple). Aggregate classes can be constructed by recursive use of these type constructors.

2.1 MOOD Overview

The general flow of execution in the MOOD system is shown in the Figure 1. MoodView [Arp 93a, Arp 93b] is the graphical user interface of the system. MoodView displays both the schema information and the query results graphically. It is also possible to update the database schema and to traverse the references in query results graphically. In displaying the schema, MoodView either makes direct calls to the Catalog Manager (1) or issues the necessary commands to the Query Manager (3). For primitive operations involving a single function call, MoodView directly communicates with the catalog. Complex operations are passed to the Query Manager. Query Manager parses and executes these commands by obtaining the necessary information from the Catalog Manager (5,6). Results are returned from the Catalog Manager or from the Query Manager depending on which subsystem received the request (2,4). Figure 2 shows the display of example schema in MoodView. Query Manager handles the method creation through Dynamic Function Linker (8, 9).

MoodView passes the MOODSQL queries to the Query Manager without any modification (3). Query Manager obtains the necessary information from the Catalog Manager (5,6) and then makes syntax and semantic checks on the queries. If no error is detected, a query tree is generated and passed to the query optimizer (7). After the optimization phase, resulting query tree is ready to be executed by the MOOD engine (10). During execution, class extents are read from the Exodus Storage Manager (ESM) and temporary results are stored in the ESM (16, 17). If the class methods are used in the query, they are activated through the Dynamic Function Linker subsystem (14,15). At the end of the execution, results are returned directly to MoodView (13). In Moodview, a user can traverse the links in the database or execute class methods with void return type by using the cursor mechanism provided.

MOOD also has a textual interface to the database. For this interface, MOODSQL provides schema definition and modification commands.

3 Query Manager

Query Manager (QM) is the main subsystem of the kernel which accepts the MOODSQL commands through the MoodView or directly from the textual interface.

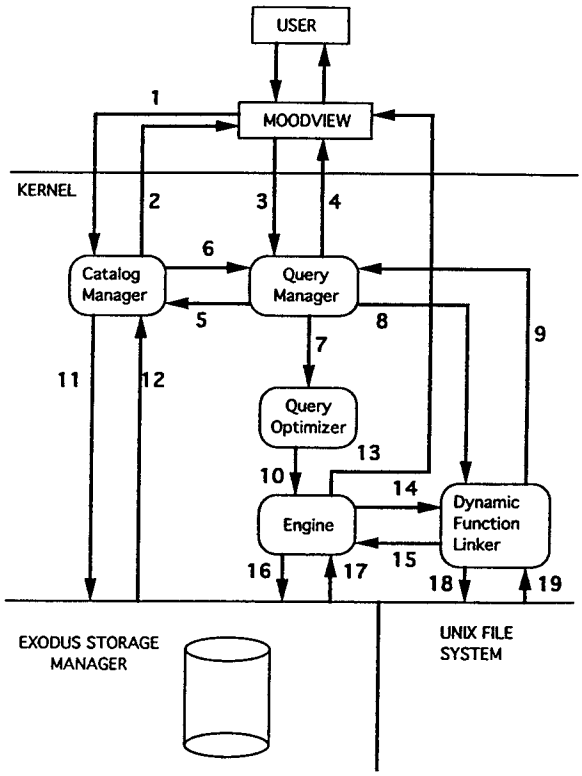

Figure 1. An Overview of the MOOD System

During the interpretation of these commands, QM communicates with the other kernel subsystems. Functionality of MOODSQL commands can be divided into two parts: schema definition and modification commands, and data manipulation commands. For schema commands, QM interacts with the Catalog Manager and in case of method definitions it also interacts with the Dynamic Function Linker. QM exports the MOOD data types and classes to the C++ compiler of the MOOD, namely MOODCC, by preparing and updating the C++ header files when a new database or a new class is created. Catalog Manager serves as the information storage of the Query Manager. For the method definitions, QM activates the Dynamic Function Linker to construct the shared object of the method and to get information about the method. For the MOODSQL queries, QM makes the syntax and semantic checks on the query using the information obtained from the Catalog Manager. It then constructs an input query tree for the optimizer by filling in the nodes with the information necessary for the optimizer and for the database engine.

A detailed description of the MOOD data model and MOODSQL is given in [Dog 94a, Dog 94c, Ozk 93a, Ozk 93b]. In the MOOD data model, the basic data

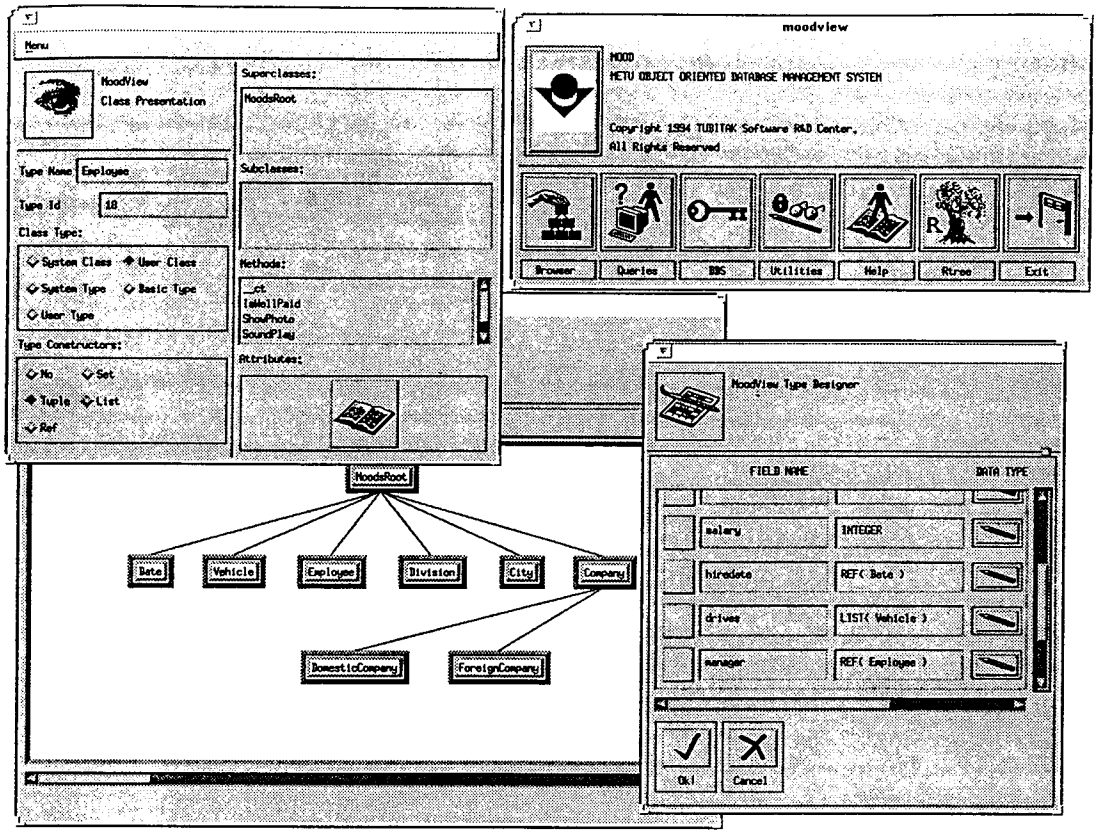

Figure 2. Display of the Example Schema

types are Integer, Float, LongInteger, String, Char, and Boolean. Any complex data type is defined using these types and by recursive application of the Tuple, Set, List and Ref type constructors. The data model also supports multiple inheritance and strongly typed methods.

The syntax of the SELECT-FROM-WHERE block is as follows:

```
SELECT projection-list
FROM   path-name r1,
       path-name r2,
       ...
       path-name rn
WHERE search-expression
```

Projection-list and search-expression may include path expressions and methods.

MOOD allows dynamic schema modifications and MOODSQL queries return objects. By using these two features a user can store the new objects obtained as the result of a query in the database. Notice that it is user's responsibility to place the new class in the inheritance hierarchy. The following example illustrates a class definition in MOODSQL.

```
CREATE CLASS Company
TUPLE(
        name            STRING[20],
        location        STRING[20],
        AnnualIncome    INTEGER,
        president       REF( Employee ),
        division        SET( Division ) )
```

An example MOODSQL query that finds the president who drives a car manufactured by his company and the company has a division at Los Angeles, is given in the following.

```
SELECT c.president.name
FROM   Company c
WHERE (c.president.drives.manufacturer = c) AND
      (c.division.location.name = 'Los Angeles')
```

The existing objects returned by MOODSQL queries can be used in populating the REF, SET and LIST attributes in UPDATE and NEW commands as illustrated in the following.

```
UPDATE Company c
SET    c.president= (SELECT e
                     FROM   Employee e
```

```
                    WHERE (e.drives.color='red')
                                and (e.ssno=2))
WHERE  c.name='METU'

NEW Company('GM','New York', 9000000,
                    SELECT e
                    FROM Employee e
                    WHERE e.ssno=24,
                    { SELECT d
                      FROM Division d
                      WHERE d.location.name =
                            'Ankara' } )
```

4 Catalog Manager

The MOOD catalog contains the definition of classes, types, and member functions in a structure similar to a compiler symbol table. The catalog is stored on the ESM. In order to achieve late binding at run time, it is necessary to carry compile time information to run time. This information is obtained from the Catalog Manager by the MoodView or by the Query Manager.

Catalog Manager uses three classes to store data definitions. These are *MoodsTypes*, *MoodsAttributes* and *MoodsFunctions* classes. The *MoodsTypes* class instances keep definitions of classes, indices and data types of the database. Instances of *MoodsTypes* class may have pointers to *MoodsAttributes* class instances which store the information about the attributes. The *MoodsFunctions* class instances keep track of the member function definitions of classes to support dynamic linking and execution of functions. Catalog classes are not any different from user defined classes. Therefore, MOODSQL can be used in accessing the required information from the catalog.

This implementation approach provides dynamic schema updates which is one of the main design considerations of MOOD. However, since versioning facility has not been implemented yet, the objects in class extents whose definitions have been modified, are deleted.

5 Dynamic Function Linker

Dynamic Function Linker (DFL) provides the late binding of methods to the objects during the query interpretation. Methods are stored as shared objects [Sun 90] in a special directory hierarchy in the UNIX file system and they are mapped to the MOOD address space if they are not already there and executed during the interpretation of MOODSQL queries. In this respect, functionality of DFL can be divided into two parts: During the definition of methods, DFL constructs shared objects and provides the necessary information about methods to the Catalog Manager through the Query Manager. During the interpretation of MOODSQL queries, DFL locates and fetches the shared objects of the methods to the memory and finds the address of a method within a shared object using the signature of the method constructed by the Query Manager. After DFL performs these operations, the Database Engine executes the method for each instance and the return value is used in the expression. In the definition phase of a method, DFL first constructs the shared object of the method. This step requires the compilation of the method body with the C++ compiler of MOOD, namely MOODCC. MOODCC is developed by modifying the cfront part of the AT&T C++ compiler (Release 2.0). The purpose of this modification is to extract the necessary information about the method, to pass the parameter values and to get the result values from the method after its execution. As a result, users can code the method bodies without any restriction and method codes are modified transparently by the MOODCC. A detailed description of the implementation of DFL is presented in [Alt 94].

MOOD also allows users to define methods returning no value (i.e. void return type). Such methods can be included in projection expressions. In this case instances are located in Database Engine but activation of the method is left to the user in the MoodView.

Users also have the chance of using the ESM calls in the methods which provides them to perform low level object operations in the methods.

6 The Query Optimizer and Database Engine

The MOOD Optimizer is implemented using the Volcano Extensible Optimizer Generator (VOG) [McK 93]. The Volcano Query Optimizer Generator is a data model independent tool that is used to develop a query optimizer for a DBMS. VOG search algorithm uses dynamic programming with branch-and-bound pruning based on cost. The Volcano generated optimizers produce the optimum execution plan when the transformation rules and support functions are provided properly. The Volcano optimizer generator uses two algebras, called the logical and the physical algebras. The job of a generated optimizer is to map an expression of the logical algebra (a query) into an expression of the physical algebra (a query evaluation plan consisting of algorithms). To do so, it uses transformations within the logical algebra and cost-based mapping of logical operators to physical algebra [McK 93]. A detailed description of the MOOD optimizer is presented in [Dur 94].

Since the paradigm of MOODSQL is the selection from extents of the classes, the traditional set and

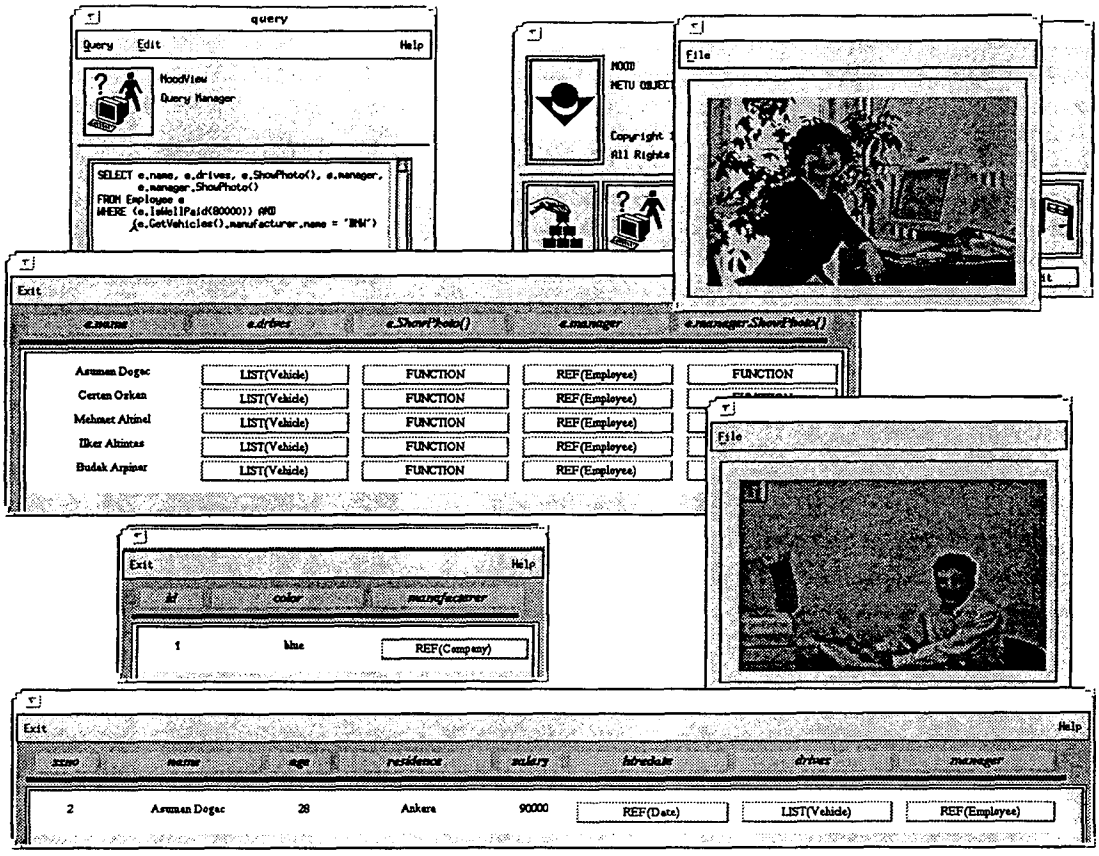

Figure 3. A MOODSQL Query and Resulting Screens

relation operators are accepted as the basis of the logical algebra. Selection, Join, Intersection, Projection and Union are defined as in the relational systems. An Unnest operator is used to manipulate set-valued components. In addition, a materialize operator (MAT) is used as defined in [Bla 93] to represent each link of path expressions such as employee.department.manager.name. The purpose of MAT operator is to indicate to the optimizer where path expressions are used so that algebraic transformations can be applied. The GET_SET operator appears at the leaves of the query tree. It includes a class into the scope of the query. Currently, the physical algebra operators implemented are as follows:

- Selection operator (FILTER)
- Indexed Selection (IND_SEL)
- Dereference operator (DEREF)
- Bind operator (BIND)
- Nested Loops Join (NESTED_LOOP_JOIN)
- Pointer Hybrid Hash Join (PRT_HH_JOIN)
- Sort-Merge Join (SM_JOIN)
- Hash Partition Join (HASH_JOIN)
- Sort operator (SORT)
- Union operator (UNION)
- Difference operator (DIF)
- Intersection operator (INT)
- Materalize operator (TRAVERSE)
- Unnest operator (UNNEST)
- Asset operator (ASSET)
- Aslist operator (ASLIST)

Database engine is the query execution subsystem of the MOOD. It consists of mainly four parts: expression interpreter generator, query interpreter, expression execution subsystem and object buffer.

At the end of the query execution, the information about the extent in which resulting objects are stored is returned to the MoodView or to the textual interface.

7 Interacting with MoodView

In displaying the query results, MoodView provides facilities to traverse the links in the database and to execute class methods with void return type. A MOODSQL query and the resulting screens in MoodView are shown in Figure 3.

Users also have the chance of accessing the MOOD Kernel from their application programs written in C++. For this purpose MOOD Kernel defines a class named UserRequest that contains a method for the execution of SQL statements.

```
class UserRequest {
        // Local variables of UserRequest
        .....
        // Method for Query Execution
        errorMessage executeQuery( ..... );
}
```

Whenever a user action requires a database operation at the schema or instance levels, user passes the corresponding SQL statements to the kernel through executeQuery method in the application program. The function returns a message indicating the success or failure of the operation. Using the cursor mechanism user can access the results of the queries.

8 Conclusions

We have left the implementation of a persistent C++ for our system as the final phase. Although persistent C++ is not implemented for the MOOD yet, the following features of the MOOD provide a powerful application development environment: MOOD kernel can be accessed from C++ application programs and there is no impedance mismatch between MOOD types and C++ types. Furthermore full C++ power is available in the methods.

References

[Alt 94] Altinel, M, "Design and Implementation of a Dynamic Function Linker and an Object Algebra for the MOOD", MS. Thesis, Dept. of Computer Eng., METU, September 1994.

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

[Arp 93b] Arpinar, B., "An Advanced Graphical User Interface for Object-Oriented DBMSs: MoodView", M.S. Thesis, Dept. of Computer Eng., METU, September 1993.

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

[Dog 94a] Dogac, A., Ozkan, C., Arpinar, B., Okay, T., Evrendilek, C., "METU Object-Oriented DBMS", *Advances in Object-Oriented Database Systems*, A.Dogac, T. Ozsu., A. Biliris, T. Sellis (Edtrs.) Springer Verlag, 1994.

[Dog 94b] Dogac, A., et al, "METU Object-Oriented DBMS", Demo description, in *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, 1994.

[Dog 94c] Dogac, A., MOOD User Manual, 1994.

[Dur 94] Durusoy, I., " MOOD Query Optimizer", M.S. Thesis, Dept. of Computer Eng., METU, February 1994.

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

[McK 93] McKenna, W. J., "Efficient Search in Extensible Database Query Optimization: The Volcano Optimizer Generator", PhD thesis, Department of Computer Science, University of Colorado, 1993.

[Ozk 93a] Ozkan, C., Dogac, A., Evrendilek, C., Gesli, T., "Efficient Ordering of Path Traversals in Object- Oriented Query Optimization", In *Proc. of Int. Sym. on Computer and Information Sciences*, Istanbul, Nov. 1993.

[Ozk 93b] Ozkan, C., " Design and Implementation of an Object-Oriented Query Language, MOODSQL, and its Optimizer", M.S. Thesis, Dept. of Computer Eng., METU, September 1993.

[Ozk 94] Ozkan, C., Dogac, A., Durusoy, I., "An Efficient Heuristics for Join Reordering", TUBITAK Software R&D Center, Tech. Rep. 94-2, January 1994.

[Sun 90] Sun Microsystems, "Shared Libraries", Programmer's Overview, Utilities and Libraries, 1990.