ALEX OBJECT-ORIENTED DATABASE MANAGEMENT SYSTEM

SOUHEIR A. FOUAD
souheir@alex.eun.eg

AMANI A. SAAD
amani@alex.eun.eg

MOHAMED G. ELFEKY
mgsaied@ritsec2.com.eg

Computer Science and Automatic Control Department
Faculty of Engineering
Alexandria University
Alexandria 21544, Egypt

ABSTRACT

This paper describes the design and the implementation of the ALEX Object-Oriented Database Management System (OODBMS). The objective of this project is to build an OODBMS that follows the standards put forward by the Object Data Management Group (ODMG). Not all the standard components are considered in the current system but will be considered by the near future. The system enables the user to define the object database model through an Object Definition Language (ODL), and to build queries about the data through an Object Query Language (OQL). Also, the system enables the user to write the object method code using the ***C++*** language, and to call these methods through the OQL. Furthermore, the system includes a graphical user interface to facilitate all the steps the user must follow to build an Object-Oriented Database System.

Keywords: Database Management System, Data Definition Language, Object Definition Language, Object Data Management Group, Object-Oriented Database Management System, Object Query Language

1 INTRODUCTION

Object-Oriented Databases combine Object-Oriented language capabilities with the storage management functions of the traditional Relational Databases. The lack of a standard for Object-Oriented Database Systems was a major limitation to their widespread use. The success of the Relational Database Management Systems did not result from a higher level of data independence or a simpler data model than previous systems, but from the standardization that they offer. That is why the Object Data Management Group put forward a set of standards for Object-Oriented Databases [1].

Our objective in this project is to build an OODBMS that follows the ODMG standards to be used as a testbed for developing and evaluating different query processing algorithms, indexing techniques, clustering techniques, and data mining tools for Object-Oriented Databases. There are a few commercial OODBMS's which support ODMG standards such as O2 and OBJECTSTORE [7]. The ODMG standard, Release 1.2, is composed of the following components:

- *Object Model* which is the common data model to be supported by any OODMBS,
- *Object Definition Language (ODL)* which is a specification language used to define the constructs that conform to the Object Model,
- *Object Query Language (OQL)* which is a declarative language for querying and updating database objects, and
- *C++ Language Binding* which explains how to write portable C++ code that manipulates the database objects. This is called the C++ Object Manipulation Language (OML).

This paper describes the design of the ALEX OODBMS. It may serve as an interesting example for an OODBMS that follows the standards of ODMG. The rest of the paper is organized as follows. Section 2 describes the Object Data Model. Section 3 describes the system design and shows the interactions between the different components of the system. Section 4 discusses the Object Definition Language and the modifications made to the grammar to facilitate the implementation of its interpreter. Section 5 shows how to integrate methods for the object types in the schema using ***C++*** and how these methods are compiled and linked to the ALEX system. Section 6 presents the Data Dictionary used to physically store the Object Schema. Section 7 describes the Object Query Language used to query the object database. Section 8 discusses the physical storage of the Database. Finally, Section 9 summarizes the paper and presents different work to be done in the near future.

2 THE OBJECT DATA MODEL

This Section describes the Object Model supported by ODMG-compliant Object-Oriented Database Management Systems. The Object Model specifies the constructs that are supported by an OODBMS:

- The basic modeling primitives are the *object* and the *literal*. Each object has a unique identifier. A literal has no identifier.
- The state of an object is defined by the values it carries for a set of *properties*. These properties can

be *attributes* of the object itself or *relationships* between this object and one or more other objects.

- The behavior of an object is defined by the set of *operations* that can be executed on or by the object.
- Objects and literals can be categorized by their *types*. An object is sometimes referred to as an *instance* of its type.
- A database stores objects. A database is based on a schema that is defined in ODL and contains instances of the types (classes) defined by its schema.

The ODMG Object Model specifies what is meant by objects, literals, types, operations, properties, attributes, relationships, and so forth. An application developer uses the constructs of the Object Model to construct the Object Model of the application. The following subsections will describe these constructs and their specification.

2.1 Types (Classes)

The definition of a *type* (*class*) has two aspects: an *interface* and one or more *implementations*. The interface defines the external characteristics of the objects of the type. These external characteristics are the *attributes* describing the state of each object of the type, and the *operations* that can be invoked on each object of the type. The implementation defines the internal aspects of the objects of the type. It consists of a set of *methods*. The method is the body of the operation defined in the interface. The separation between the interface and the implementation is the way that the ODMG Object Model reflects *Encapsulation* [2].

The ODMG Object Model supports *Inheritance* relationships between classes; i.e., a class, called the sub-class, inherits the same interface of another class, called the super-class. The sub-class may add to the interface and may have another implementation to the same operations of the super-class. Hence, the ODMG Object Model supports *Polymorphism*.

A *type* may have an *extent* which is the set of all its instances (objects). Also, it may have one or more *keys* which uniquely identify a specific object. The key is either simple, consisting of a single property, or compound, consisting of a set of properties.

2.2 Objects

An object is an *instance* of a class. It has an identifier to be distinguished from other objects, a name used by developers or users to refer easily to that object, a lifetime to determine how the memory and the storage allocated to objects are managed, and a structure which may be atomic or not. There are two lifetimes supported by the ODMG Object Model: *transient* and *persistent*. A transient object is allocated memory by the programming language run-time system when it is declared in a method, and the memory is deallocated after the method returns. On the other hand, a persistent object is allocated memory and physical storage by the OODBMS run-time system, and continues to exist on the physical storage. Hence, persistent objects are sometimes referred to as *database objects*.

2.3 Literals

Literals do not have identifiers. The ODMG Object Model supports three literal types: atomic, collection and structured. Atomic literals include the basic data types such as *Long*, *Short*, *Double*, *Boolean*, *Char*, *String*, *Enum*, … etc. Collection literals include *Set*, *Bag*, *List* and *Array*. Structured literals include *Date*, *Interval*, *Time*, and *Timestamp.* Also, the ODMG Object Model supports the user-defined structures.

2.4 State and Behavior

The *state* of an object is determined by the values of its properties. A property of an object is either an *attribute* or a *relationship*. An attribute is defined by its name and its type which is in turn atomic, a collection or structured. A relationship is defined between two types that an object of one type refers to one or more objects of the other type. The reference is made by the object identifiers. The ODMG Object Model supports only binary relationships. A relationship has a name and a reverse path containing the name of another relationship defined elsewhere. In the definition of the later relationship, the reverse path must contain the name of the former relationship.

The *behavior* of an object is specified as a set of *operation* signatures. Each signature defines the operation name, the name and the type of each argument, and the return value type. In Object-Oriented terminology, we refer to the implementation of an operation as a method. Thus, an object type is defined by a set of attributes and a set of relationships (constituting its state), and a set of methods (constituting its behavior).

2.5 Example

Figure 1 presents an object-oriented database schema using the *class diagram* of the Unified Modeling Language (UML) notation.

This object-oriented database schema contains three classes: “Person”, “Employee”, and “Department”. “Employee” is a subclass of “Person” and so there is an *inheritance* relationship from “Employee” to “Person”. There are two bidirectional relationships between “Employee” and “Department”, one indicates that a “Department” instance has many “Employee” instances working into, and the other indicates that a “Department” is managed by one and only one “Employee”.

3 THE ALEX SYSTEM DESIGN

This Section describes the ALEX system design and shows the interactions between its different components. Clearly, as any DBMS, there are multiple levels of users of the system.

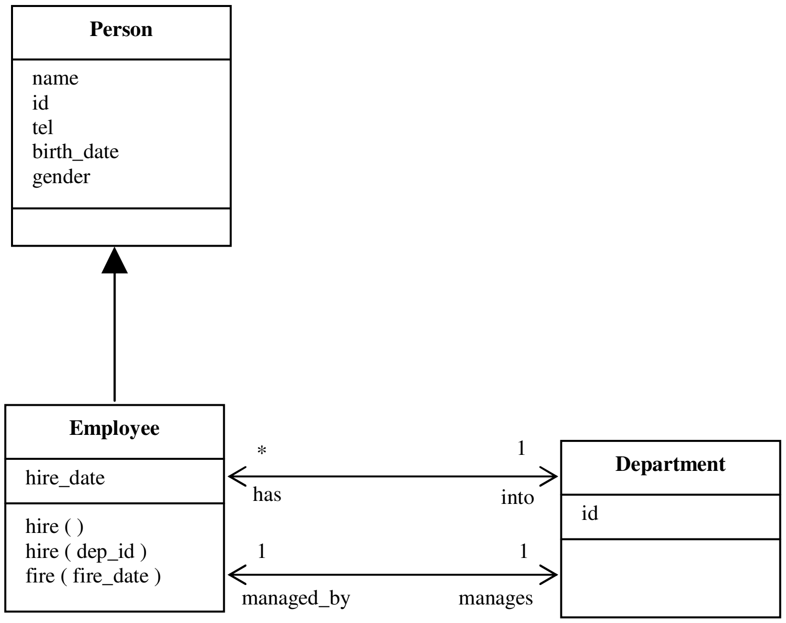

Figure 1: CLASS DIAGRAM FOR AN OBJECT-ORIENTED DATABASE SCHEMA

The first level is the developer who builds the database application, and the second is the user of the application who manipulates the data using queries.

Figure 2 shows the two main components of the entire system. They are the ALEX system and the ALEX Dynamic Linking Library (DLL). The ALEX system is used by the developer to build an Object-Oriented Database model. The Object Schema is defined using either the Graphical User Interface (GUI) or the ODL. Also, the end user manipulates the database using OQL statements. The developer must also write the file containing the ***C++*** code of the object types methods and supply it to the system. The ALEX system automatically generates two other files depending on the Object Schema specified.

Those three files are compiled using a ***C++*** compiler to generate the ALEX DLL. The ALEX DLL is the library containing the object code of all the methods of the object types in the schema to be dynamically linked to the ALEX system at run time.

Figure 3 shows all the components of the ALEX system. The user defines the Object Schema either by the *Graphical Interface* or by writing an ODL file interpreted by the *ODL Processor*. The Object Schema definition is passed to the *Schema Manager*. The *Schema Manager* builds this Object Schema by storing it in the Data Dictionary and also generates the ***C++*** header file and the DLL main file.

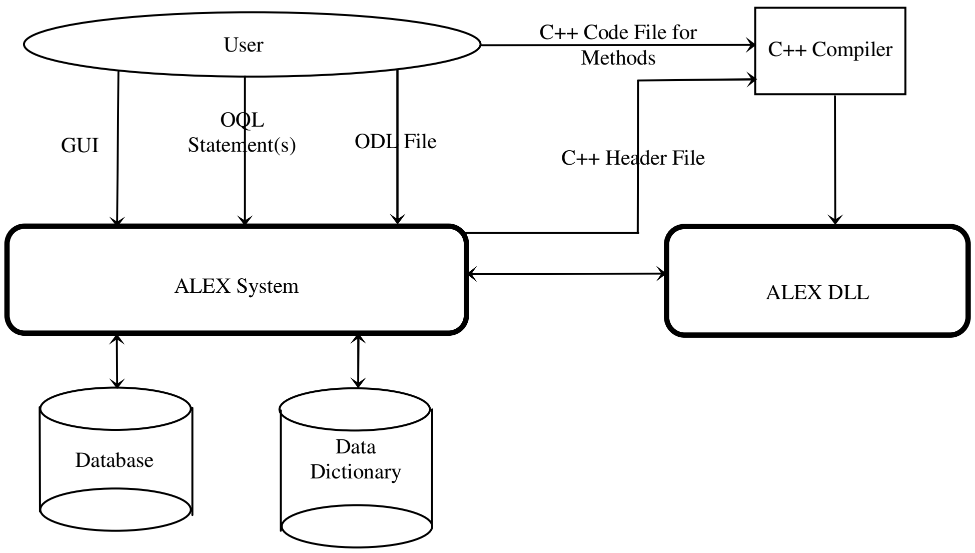

Figure 2: THE MAIN COMPONENTS OF ALEX

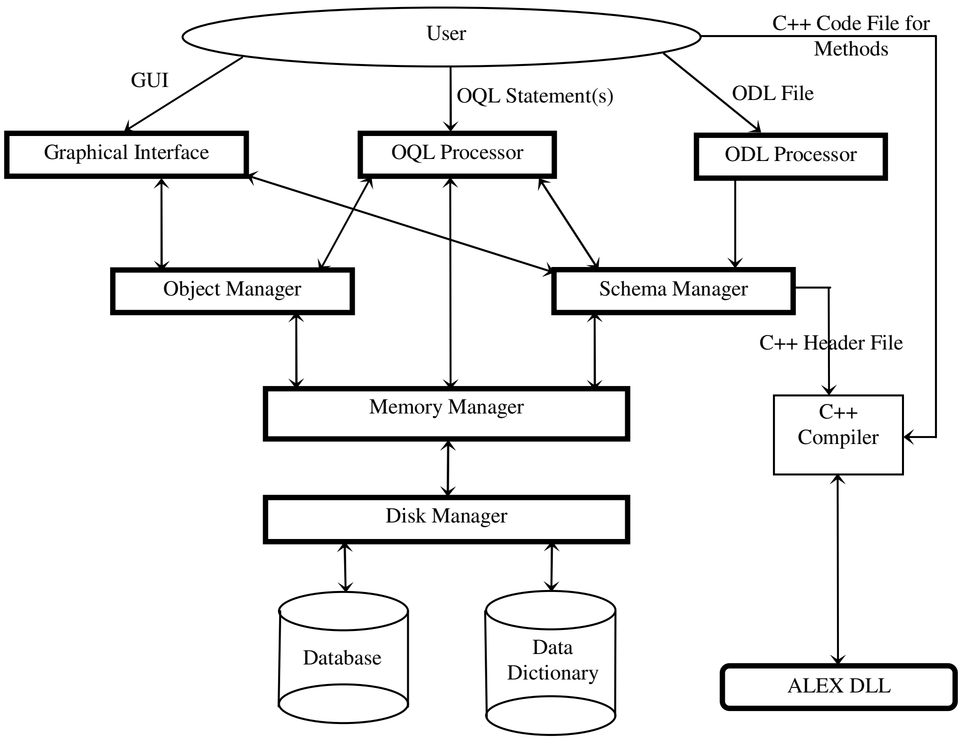

Figure 3: ALL THE COMPONENTS OF ALEX

The user may deal with objects, creating new objects or querying the current objects data, using the *Graphical Interface* or writing OQL statements interpreted by the *OQL Processor*. The *Object Manager* is the component that is responsible for dealing with objects data through the *Memory Manager*. The *Memory Manager* is responsible for dealing with the Data Dictionary to retrieve the desired information required by other components. It holds some of the most used information, such as the inheritance graph, in memory. Also, it is responsible for dealing with object data stored in the object database. It uses the *Disk Manager* intensively to handle the files on disk.

4 OBJECT DEFINITION LANGUAGE

The Object Definition Language (ODL) is a specification language used to define the interfaces of the object types that conform to the ODMG Object Model. The ODL supports all semantic constructs of the ODMG Object Model. It is programming-language independent and is not intended to be a full programming language.

The ODL is a Data Definition Language (DDL) for the object types. It defines the characteristics of the types including their properties and operations. The ODL defines only the signatures of the operations and does not address the definition of the methods that implement those operations. Each sentence of the ODL defines a type that is either an ***interface*** (the abstract definition of an object), a ***type*** (a structure or an enumeration) or a ***constant***.

The complete BNF for the ODL is given in [1]. Some modifications are made to this grammar so that it can be implemented using a recursive descent parser [3]. These modifications are *left factoring* and *elimination of left recursion* described in [3].

Examples:

left factoring

Rule (8) :

Old : <interface_body> ::= <export> | <export> <interface_body>

New : <interface_body> ::= <export> <X8>
<X8> ::= ∈ | <interface_body>

elimination of left recursion

Rule (17) :

Old : <and_expr> ::= <shift_expr> | <and_expr> & <shift_expr>

New : <and_expr> ::= <shift_expr> <X17>
<X17> ::= ∈ | & <shift_expr> <X17>

The parser passes through the ODL file checking its syntax and semantics. It extracts all the required information about the Object Schema to pass it to the *Schema Manager*. This information includes, for each sentence, the *definition type* (***interface***, ***type***, or ***constant***) and the *definition name*.

For the *interface definition*, the information includes also the *interface parents*, *keys*, *lifetime* (***persistent*** or ***transient***), *attributes* and the *data type* of each one, *operations* and the *return data type* of each one, *parameters* of each *operation* and the *data type* of each *parameter*, and *relationships* and the *reverse path* of each one. For the *type definition*, the parser checks whether this *type* is *structure* or *enumeration*. For the *structure type definition*, the parser extracts its *members* and the *data type* of each one. For the *enumeration type definition*, the parser extracts its *enumerators*. For the *constant definition*, the parser extracts its *data type* and *value*.

The parser handles a symbol table to prevent duplication names in interfaces, types and constants. Also, it prevents the duplication names in attributes of one interface and the duplication operation signatures of one interface. The parser also handles a relationship table to ensure that each relationship is defined correctly. An example of an ODL file is shown in Figure 4.

```
typedef struct DateType
{
        Short day ;
        Short month ;
        Short year ;
} Date ;

enum Gender { Male , Female } ;

interface Person
(
        extent people
        key ( name , id )
)
{
        attribute String name ;
        attribute Long id ;
        attribute Struct Telephone { String no ; Enum Type { Home ,
                                     Work , Mobile } type ; } tel ;
        attribute Date birth_date ;
        attribute Gender gender ;
};

interface Employee : Person
()
{
        attribute Date hire_date ;
        void hire ( ) ;
        void hire ( in Unsigned Short dep_id ) ;
        boolean fire ( out Date fire_date ) ;
        relationship Department manages
                inverse Department :: managed_by ;
        relationship Department into
                inverse Department :: has ;
};

interface Department
(
        key ( id )
)
{
        attribute Unsigned Short id ;
        relationship Employee managed_by
                inverse Employee :: manages ;
        relationship Set<Employee> has
                inverse Employee :: into ;
};
```

Figure 4: AN EXAMPLE OF AN ODL FILE.

5 INTEGRATING METHODS

As outlined before, The object behavior is a main issue in Object-Oriented modeling. The object behavior is modeled using operations. A method is the implementation for an operation of a specific object type.

The problem about allowing the user to write the object methods is that, when the user modifies one of the methods or modifies the Object Schema by deleting or adding new methods, the system must dynamically link the new methods, relink the modified methods, and remove the deleted methods. In order to accomplish that, the system must contain a linker. Although it is the most efficient solution, it is hard to be implemented as it requires a detailed study of the operating system.

The technique we used to support methods in our system is to generate a dynamic linking library (ALEX DLL) containing all the methods of all the object types of the current schema. This library is based on the concept of the Dynamic Linking Library (DLL) introduced by *Microsoft Windows*. The DLL functions are linked only when they are called from another application. When a function in the DLL is modified, all what is done is the recompilation of that DLL, and hence, the application is independent of the version of the function it calls.

Clearly, any application must be independent of the implementation of the functions it calls. Thus, the ALEX system must be independent of the name of the object type and the names of its methods; i.e., independent of the Object Schema which may be modified by the user anytime. Thus, the ALEX DLL, which contains all the methods of the object types in the Object Schema, will contain only one function to be called from the ALEX system. This function, called the *DLL main function*, is the one that is responsible for calling the required method from the required object type. The input parameters of this function are the object type name, the object name, the method name and a list of the values of the parameters of this method.

When the user defines a new schema, the *Schema Manager* generates automatically three files written in ***C++***. They are (i) a header file defining the classes and the data types of the schema, (ii) a skeleton source code file ready for the user to write the body of the methods, and (iii) the main file of the DLL which contains the DLL main function. Figure 5 shows an example of those files generated for the example in Figure 4.

Now that the ALEX DLL system, which consists of the header file, the code file and the main file, is ready for compilation, it is compiled and the DLL file is generated. If the user modifies any method or modifies the Object Schema, the ALEX system modifies the header file, the code file and the DLL main function, and the ALEX DLL must be recompiled.

The ALEX system is now ready for calling any object method. It calls the *DLL main function* providing the class name, the object name, the method name and a list of the values of its parameters. This function calls the appropriate method and returns to the ALEX system the same value returned from the method.

By this approach, the ALEX system is independent of the Object Schema and the code of the methods.

```
// File : SampleSchema.h
// Created by : ALEX OODBMS
// Description : The header file of the schema.
// Date : Friday, 11 December, 1998
// Contents : The classes and the data types definitions.

typedef struct DateType
{
        short day ;
        short month ;
        short year ;
} Date ;

typedef enum Gender { Male , Female } ;

class Person
{
  // Attributes
  CString name ;
  long id ;
  struct Telephone
  {
    CString no ;
    enum Type { Home , Work , Mobile } type ;
  } tel ;
  Date birth_date ;
  Gender gender ;

  // Operations

  // Relationships
} ;
typedef Set<Person> people ;

class Employee : Person
{
  // Attributes
  Date hire_date ;

  // Operations
  void hire ( ) ;
  void hire ( unsigned short dep_id ) ;
  BOOL fire ( Date &fire_date ) ;

  // Relationships
  Department manages ;
  Department into ;
} ;

class Department
{
  // Attributes
  unsigned short id ;

  // Operations

  // Relationships
  Employee managed_by ;
  Set<Employee> has ;
} ;
```

Figure 5a: THE HEADER FILE.

```
// File : SampleSchema.cpp
// Created by : ALEX OODBMS
// Description : The code file.
// Date : Friday, 11 December, 1998
// Contents : The skeleton of the methods to be completed by the user.

#include "SampleSchema.h"

void Employee :: hire ( )
{
}
void Employee :: hire ( unsigned short dep_id )
{
}
BOOL Employee :: fire ( Date &fire_date )
{
}
```

Figure 5b: THE CODE FILE.

```
// File : DLLmain.cpp
// Created by : ALEX OODBMS
// Description : The DLL main file.
// Date : Friday, 11 December, 1998
// Contents : The DLL main function.

#include "SampleSchema.h"

BOOL InvokeMethod ( CString ClassName , CString ObjectName ,
        CString MethodName , void* Parameters , void* ReturnValue )
{
  if ( ClassName == "Person" )
  {
    return False ;
  }
  else if ( ClassName = = "Employee" )
  {
    object = GetObject ( ObjectName ) ;
    if ( MethodName = = "hire" )
    {
      if ( Parameters )
      {
        object.hire ( Parameters[0] ) ;
        return True ;
      }
      else
      {
        object.hire ( ) ;
        return True ;
      }
    }
    else if ( MethodName = = "fire" )
    {
      object.fire ( Parameters[0] ) ;
      return True ;
    }
  }
  else if ( ClassName = = "Department" )
  {
    return False ;
  }
  else { return False ; }
}
```

Figure 5c: THE DLL MAIN FILE.

6 DATA DICTIONARY

The Data Dictionary is a file in the secondary storage to describe the Object Schema of the DBMS. It is used by the OODBMS in structuring databases and at run time to guide access to databases.

There are two alternatives to implement the data dictionary. The first is to design a new format for this file; the second is to use a well-known format. Clearly, the second alternative is better especially when using a well-known DBMS file format and dealing with it through ready-made functions supporting Call-Level Interface

(CLI). We chose *Microsoft Object DataBase Connectivity (ODBC)* which contains the operations to access data in any database having an ODBC driver. These operations include handling tables and records, and transaction management. They allow any application to be independent of the DBMS used. We chose here *Microsoft DataBase (MDB)* file format for two reasons. The first is its widespread use and the second is that it gives us the facility to change the database file name dynamically at run time. Clearly, the ALEX system needs to change the database file name at run time in order that one user may transfer from one Object Database Model to another one at the same session.

Figure 6 illustrates the Extended Relationship Diagram (ERD) of the data dictionary tables.

recursive descent parser [3]. These modifications are *left factoring* and *elimination of left recursion*.

The parser passes through the OQL statement to check its syntax and semantics. It extracts all the required information about the query and passes it to the query processor which is responsible for executing the query. First of all, the parser determines the query type. The parser distinguishes four main query types: ***literal***, ***construction***, ***aggregate***, and ***selection***. A *literal query* is a constant value (numerical, string, character, or enumeration). A *construction query* constructs an object or a type (structure or collection). *Construction queries* are either *transient* (for types and transient objects) or *persistent* (for persistent objects). An *Aggregate query* contains an aggregate function to be calculated. A *selection query* is a *select-from-where* SQL-like query.

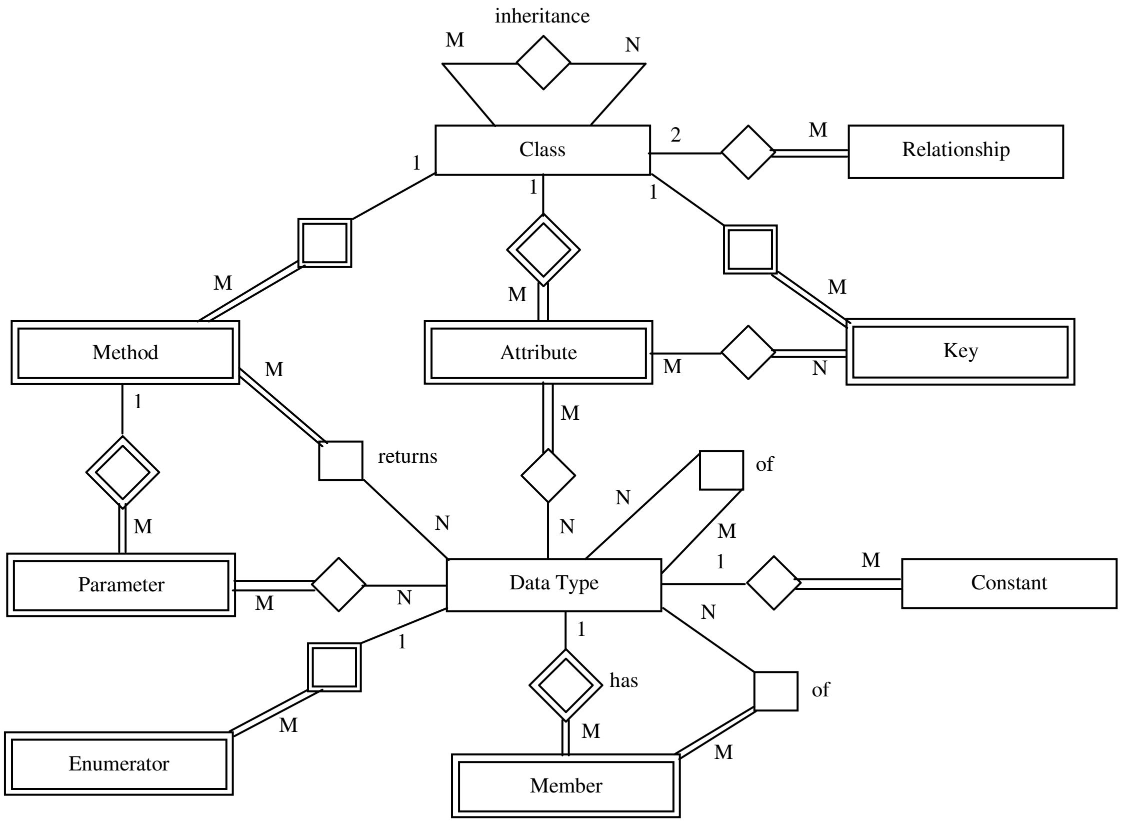

Figure 6: THE ERD OF THE DATA DICTIONARY

7 OBJECT QUERY LANGUAGE

The Object Query Language (OQL) is a complete and simple language used to query the object database of the ALEX system. It supports the ODMG data model. It deals with complex objects without privileging the set construct and the *select-from-where* clause. The design principles, the main features and the complete BNF of this language are fully described in [1].

As in the ODL, some modifications are made to the grammar so that it can be implemented by using a

The query processor executes the query according to its type. For the *literal queries*, the query processor returns the literal value. For the *construction queries*, it checks the type to be constructed whether it is *persistent* or *transient*, and if it is *persistent*, the query processor creates this object in the physical storage and returns the Object Identifier (OID) value created for that object. If the type to be constructed is *transient* (set, list, bag, array, structure or user-defined type), the query processor returns a value according to its type. For the *aggregate queries* (count, sum, min, max, … etc), the query processor evaluates the specified function and returns the evaluated value.

For the *selection queries*, the query processor retrieves the object identifiers of all the objects of the classes specified in the *from* clause, and then filters these objects according to the *where* conditions, and finally projects only the required properties according to the *select* clause. The *selection query* is written in the same way as the selection query in the traditional SQL (*select-from-where* clause). Also, OQL supports the *path expressions* allowed in the object-oriented languages. A *path expression* uses the "." or "->" notation to navigate a complex object to get its internal properties.

Figure 7 shows some examples for OQL statements for the Object Schema of Figure 4. Those examples outline the various features of OQL.

```
Employee ( name: "Ahmed" , id: 123123
           , tel: Telephone ( no: "596-0330" , type: Home )
           , birth_date: Date ( day: 29 , month: 8 , year: 1976 )
           , gender: Male , into: Department ( id: 1 ) )

First ( Select e From Employee e Where e.name = "Ahmed" )

Select e.name From Employee e , Department d
Where ( e.into = d ) And ( d.id = 1 )

Select e.name From Employee e Where e.into.id = 1

Select e.name From Employee e Where e.birth_date.year > 1970
```

Figure 7: SOME EXAMPLES OF OQL STATEMENTS.

The first OQL statement constructs a new object of the class *Employee* having the specified properties. When the query processor executes this statement, it will recognize 13 queries (since the OQL BNF is recursive). They are *"Ahmed"* (literal), *123123* (literal), *"596-0330"* (literal), *Home* (literal), *Telephone(…)* (transient construction), and so on. The last recognized one is *Employee(…)* (persistent construction). Note that the query *Department (id: 1)* is a persistent construction query and so the query processor creates a new object of the class *Department* in the physical storage in addition to the new object of the class *Employee*. Note also that the query processor takes care of the relationships between classes, and so, the query processor will update the value of the property *has* for the created *Department* to contain the object identifier value of the created *Employee* since *Department::has* is the inverse traversal path of the relationship *Employee::into*.

The second statement selects the first element in the *Employee* class having name "*Ahmed"*. This statement shows how to combine an aggregate function *first* with a selection query. This simple selection query returns the object identifier value of the specified object. The third statement retrieves the names of the employees in the department whose id is 1 using an explicit join. The fourth statement shows how to write the same query using a path expression which represents an implicit join which is an advantage of the object-oriented data model. The last one retrieves the names of the employees born after 1970. It shows that a path expression may contain a property of a class as well as a property of a user-defined structure.

8 PHYSICAL STORAGE

There are two standard approaches to physically store the object-oriented database. (i) to map it to an underlying relational database system, or (ii) implement an advanced storage server that offers more complex structures than flat relations [9].

Here, we considered the first approach since it offers the advantage that one could rely on established, matured and portable technology. On the other hand, it is unlikely to obtain a good performance since the complex structures of the object-oriented database schema have to broken into small pieces in order to be stored in flat relations. Hence, queries to the object-oriented database have to be mapped into large joins queries to the relational database. Although the second approach has a superior performance due to the flexible physical database organization that allows for efficient query processing, it has the obvious disadvantage that one has to implement a new storage manager with more powerful capabilities for complex structured data, which is intended for the future work in the ALEX system but is not currently available.

The relational mapping of the object-oriented database contains two main relations, one for the objects and the other for the values of the properties of the objects. Just like the data dictionary, the relational database used is *Microsoft DataBase*, and dealing with it is done through *Microsoft Object DataBase Connectivity (ODBC)*. The ERD of these relations is shown in Figure 8. Note that OID stands for the object identifier.

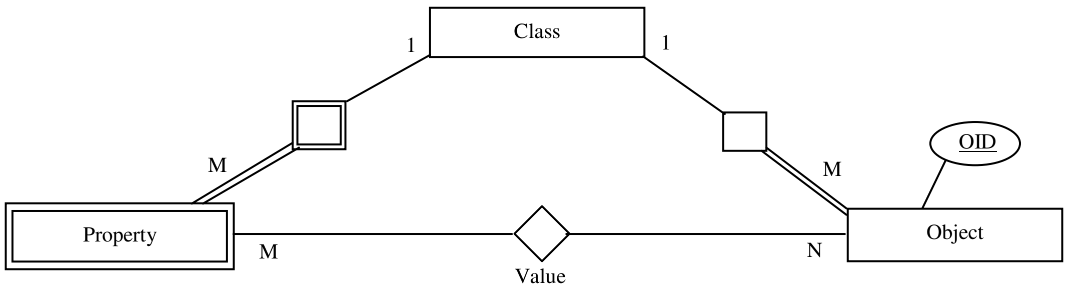

Figure 8: THE ERD OF THE DATABASE

9 SUMMARY AND FUTURE WORK

In this paper, we described the main issues in the design of an OODBMS called ALEX that follows the standards of ODMG, Release 1.2. We illustrated the variant system components design. We presented the implementation issues encountered in designing the ODL parser, OQL parser, and the Data Dictionary. We investigated a new approach to easily integrate methods into the Object-Oriented Database system.

In the future, a query optimizer can be added to the system above the query processor to optimize the query execution which is still an open area for research in Object Oriented Databases. Furthermore, to be a full DBMS, ALEX should include *Recovery Management*, *Transaction Management* and *Concurrency Control* techniques. Hence, we will add these important features to ALEX by the near future.

Data Mining techniques are very rare for Object-Oriented Databases, and so we will consider extending the features of ALEX to support data mining. We are now in the process of developing a data mining query language for Object-Oriented Databases.

ACKNOWLEDGEMENT

We are grateful to Dr. Walid G. Aref who guided us through all the phases of this work supplying us by references and, more importantly, advises. Also, we wish to express our particular thanks to Ghada M. Badr who participated in the implementation of the system.

REFERENCES

[1] R.G.G. Cattell. *The Object Database Standard: ODMG-93* (Release 1.2). Morgan Kaufmann Publishers, Inc. San Francisco, California, 1996.

[2] E. Bertino and L. Martino. *Object-Oriented Database Systems*. Addison-Wesley Publishing Company, 1994.

[3] Alfred V. Aho, Ravi Sethi, and Jefrey D. Ullman. *Compilers Principles, Techniques, and Tools.* Addison-Wesley Publishing Company, 1986.

[4] Amani A. Saad, and Ghada M. Badr. The ALEX Object Manager. In *Proceedings of ISCC'97: Second IEEE Symposium on Computers and Communications*, pages 200-204, Alexandria, Egypt, July 1997.

[5] Mohamed G. Elfeky, and Ghada M. Badr. *ALEX Object-Oriented Database Management System.* Computer Science and Automatic Control Department, Faculty of Engineering, Alexandria University. Technical Report, 1997.

[6] F. Bancilhon, C. Delobel, and P. Kanellakis. *Building an Object-Oriented Database System: The Story of O2*. Morgan Kaufmann, 1992.

[7] Peter Kueng. Comparison of ten OODBMS's. *The Swiss Computer Magazine OUTPUT*, pages 60-63, June 1994.

[8] M. Kifer, W. Kim, and Y. Sagiv. Querying Object-Oriented Database. In *Proceedings of 1992 ACM-SIGMOD Int. Conf. Management of Data*, pages 393-402, San Diego, California, June 1992.

[9] Marc H. Scholl. Physical Database Design for an Object-Oriented Database System. In Johann C. Freytag, David Maier, and Gottfried Vossen, editors, *Query Processing for Advanced Database Systems*, pages 420-447, Morgan Kaufmann, 1994.

[10] J. Han, Y. Fu, W. Wang, K. Koperski, and O. Zaiane. DMQL: A Data Mining Query Language for Relational Databases. In *1996 SIGMOD'96 Workshop on Research Issues on Data Mining and Knowledge Discovery (DMKD'96)*, Montreal, Canada, June 1996.