

QUERYING OBJECT-ORIENTED DATABASES FROM
C++

By

XIJIAN TANG

Bachelor of Science

in Petroleum Engineering

Petroleum Institute of Eastern China

Dongying, P. R. China

1983

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
MASTER OF SCIENCE
December, 1998

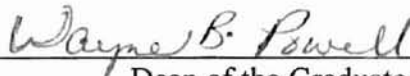
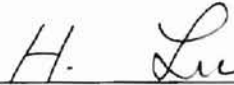
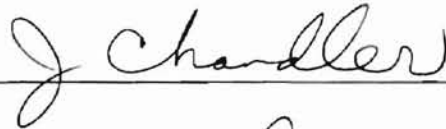
QUERYING OBJECT-ORIENTED DATABASES FROM

C++

Thesis Approved:



Thesis Adviser



Dean of the Graduate College

ACKNOWLEDGMENTS

I would like to express my sincere appreciation to my advisor Dr. George Hedrick, for his intelligent guidance, inspiration, and friendship during this study, and for his patience in correcting this thesis. My appreciation is also extended to my other committee members Dr. John Chandler and Dr. Huizhu Lu, for their kind suggestions, assistance and friendships regarding my project.

Special thanks are given to Thomas L. Underwood and all my other personal friends, for their encouragement, consideration, and helping me with my language.

I also like to express my thanks to my mother, Xiuying Feng, and to my wife, Weihua Liu for their encouragement, loves and supports during my study at Oklahoma State University.

TABLE OF CONTENTS

<u>Chapter</u>	<u>Page</u>
I. INTRODUCTION.....	1
Motivation.....	2
Problem Statement.....	4
Outline of the Thesis.....	5
II. LITERATURE REVIEW	7
Query Languages	7
Queries Integrated with the Database Programming Languages	9
The Recent Advances in Object-oriented Databases with Object Query Languages	11
III. OBJECTS AND OBJECT-ORIENTED DATABASES.....	13
Object Orientation Concepts.....	13
Object-oriented Database Concepts	14
An Application Example of Object-oriented Database.....	17
IV. A QUERYING METHOD FOR OBJECT-ORIENTED DATABASES.....	20
Introduction.....	20
Extension to C++ to Cover the Object Query Language	21
Basic Structure.....	22
Structure Construction	24
Selection Predicate.....	24
Aggregate Operators	25
Relational and Logical Operators	25
in and !in Operators	26
Nested Queries.....	27
Join Query	27
Duplicates and Order	27
The Object Query Language Grammar.....	28

<u>Chapter</u>	<u>Page</u>
V. IMPLEMENTATION OF THE QUERYING OPERATIONS FOR OBJECT-ORIENTED DATABASES.....	30
Implementation Schema of An Object-oriented Database.....	30
Object Class and Data Classes	33
Reference Classes	37
Collection Template Classes.....	39
Database Structures and System Classes	44
Design and Implementation of the Query Method.....	46
Test Results.....	52
VI. SUMMARY, CONCLUSION AND SUGGESTED FUTUTE WORK.....	60
Summary	61
Conclusions.....	62
Suggested Future Work.....	62
REFERENCES.....	64
APPENDIX.....	67
Glossary of Terms	68
Table of Acronyms	72

LIST OF TABLES

<u>Table</u>	<u>Page</u>
1. The Relational Operators	26
2. The Query Results of the Collection, people	55
3. The Query Results of the Collection, students.....	56
4. The Query Results of the Collection, staff.....	56
5. The Query Results of the Collection, TAs.....	57
6. The Query Results of the Collection, professors	57
7. The Query Results of "People Who Live in Stillwater"	58
8. The Query Results of "TA in Department 1"	58
9. The Query Results of "Professors Whose Ages Are Greater Than the Average in Department 1"	59

LIST OF FIGURES

<u>Figure</u>	<u>Page</u>
1. Using an ODBMS	16
2. Data Class Hierarchy in the University Personnel Database.	18
3. E –R Schema.....	31
4. The Tabular Representation.	32
5. Schema Definition of the Class d_Object.....	33
6. Schema Definition of the Class Person.....	34
7. Schema Definition of the Class Student.	35
8. Schema Definition of the Class Staff.....	35
9. Schema Definition of the Class TA	36
10. Schema Definition of the Class Professor	36
11. Schema Definition of the Template Class d_Ref.....	37
12. Schema Definition of the Class d_Ref_Any.....	38
13. Schema Definition of the Template Class Tstruct	39
14. Schema Definition of the Class d_Iterator.....	40
15. Schema Definition of the Class d_Collection.....	41
16. Schema Definition of the Template Class d_Set.	42
17. Schema Definition of the Template Class d_Bag.	42
18. Schema Definition of the Template Class d_List.	43

<u>Figure</u>	<u>Page</u>
19. Schema Definition of the Template Class d_Varray	44
20. Schema Definition of Data Store Structure.....	45
21. Schema Definition of the Class d_Transaction.....	45
22. Schema Definition of the Class d_Database.....	46
23. Three Symbol Tables.	48
24. Schema Definition of the Class string.	49
25. Schema Definition of the Class queryclass.....	49
26. Schema Definition of the Class exequery.....	51
27. Implementation Definition of Query Execution Function	52
28. The Main Menu.....	53
29. The Sub Menu.....	54

CHAPTER I

INTRODUCTION

Many programmers design and implement applications using object-oriented programming languages, such as C++, Smalltalk, Java. The code developed and implemented by using object design techniques tends to be well modularized. This modularization is helpful in constructing many manageable, relatively small, source-code building blocks. These software blocks, or objects, can be combined into functionality and specialized to develop new functionality. The techniques of object-oriented programming help to reduce the conceptual gap that the designers have in modeling real world problems. Object-oriented programs are easier to maintain, to modify, and to extend than other programs that were written using conventional techniques. This flexibility allows programmers to respond more rapidly to changes in new requirements.

Since the inception of object-oriented programming, object-oriented concepts have been integrated into database management systems. Database management systems (DBMS) allow persistent databases to be shared by many users concurrently. DBMS use underlying concurrency control, software management, and optimization strategies to achieve this efficiently. There are two trends for object-oriented concepts being integrated into database management systems. One trend is the incorporation of powerful

object capabilities. The object-oriented techniques provide powerful modeling and application development alternatives for a number of advanced database applications. The other trend in database is the downsizing of equipment in the environment in which database management system operates. In many corporations the internetworked LANs with file, database, and other servers are replacing mainframe environments for sharing information concurrently.

Traditional database systems cannot meet the needs of new environments that utilize new techniques on more complex applications, such as network, multimedia and heterogeneous client /server, etc. An object-oriented DBMS provides persistence for objects rather than for tables or records as do relational databases. It is well-suited for supporting multi-user applications developed with an object-oriented programming language, where the users must share data. The combination of object-oriented programming with an object-oriented database offers benefits of a synergistic development environment.

Motivation

As in all other kinds of databases, object-oriented databases must be able to respond to a query easily. Frequently, this can be achieved by using an object-oriented programming language [Loomis, 1994]. Because of the close coupling of the object-oriented database with an object-oriented programming language, application programmers can query persistent and transient objects in a consistent way. Also, object

query languages can access the object-oriented databases in the same manner as SQL is used to query relational databases.

A variety of object query languages have been proposed in the literature.

Basically these querying languages can be placed into one of two families, each family having its own objectives. The first family is the integrated family. Its objective is seamless integration with its host object programming language. One approach is that queries are coded in standard C++ using functions. The second family is the SQL family. Its objective is to modify SQL to accommodate object databases. These query languages came from the relational DBMS and have been extended to accommodate some aspects of objects, such as OQL. OQL is Object Query Language defined by the Object Database Management Group (ODMG).

These object query languages have some problems: how should programming language features be mixed with the capabilities of Object Database management Systems [ODBMS]. For example, OQL semantics of ODMG - 93 [Cattell, 1996] support querying any type of object supported by the object-oriented programming language. A query can include operations defined on those types. Another problem arises since the operations specified for classes are equally valid on both persistent and transient objects. This is quite different from the SQL-derivative perspective, where the relational DBMS provides only the collection, Boolean, aggregate, sorting, and grouping functions. All of the other functions must be provided by the programming languages. There is a clear distinction between a programming language with transient data and SQL with persistent data. SQL functions, including stored procedures, can execute only on persistent data.

They are the only database operations possible where SQL is used. The programmer continually must be aware of the differences between transient and persistent data. ODBMS tries to remove this distinction[Loomis, 1994].

Problem Statement

This topic of thesis is querying object-oriented databases using extended C++. The approach belongs to the integrated family. The reasons we choose the approach rather than the SQL approach are as follows:

1. There is a clear distinction between database variables used by SQL and program variables. Values must be copied between these two kinds of variables explicitly, and there may not be adequate type checking performed in those operations.
2. SQL has its own notions of expressions and functions. These functions cannot be used freely in object-oriented programming languages, nor can the constructs of programming languages be used in composing SQL statements.
3. SQL implements a table (relational) model, which is not the same as an object-oriented model supported by the object-oriented programming languages. The result of a SQL query is a relation which cannot be manipulated appropriately by object-oriented programming languages.
4. The application programmer must design and code the logic to deal with the differences between the type systems of the database programming language and SQL. This can be error-prone and labor-intensive.

The goal of this thesis is to find the seamlessness of the object-oriented database interface by how natural it is for the typical C++ programmer, rather than by how it fits with a standard C++ compiler. Based on the square syntax of collections presented by Loomis [Loomis, 1994] and drawing from features of either OQL or SQL, there is a new querying method for object-oriented databases. The method extends the C++ language to cover part of the object querying language. The extension to C++ is simple and easy to implement. However, the querying capability is so powerful that it effects the most queries implemented by OQL or SQL. Unlike the approach presented by Loomis, this new approach does not require use of a preprocessor that turns the new syntax into code that the compiler can understand. Instead, there is a query execution function to take the queries as its arguments. This extended C++ approach has the benefits of an easy-to-use interface with run-time type checking.

Outline of the Thesis

The thesis is organized as follows: Introduction, Literature Review, Objects and Object-oriented Databases, A Querying Method for Object-oriented Databases, Implementation of the Querying Operations for Object-oriented Databases, and Summary, Conclusion, and Suggested Future Work. Introduction simply describes the current development of object-oriented database and object query problems. Literature Review introduces several querying database approaches or/and languages such as SQL and OQL. Objects and Object-oriented Databases first reviews the concepts of objects and object-oriented databases, then describes object database organization and gives an example of object database. A Querying Method for Object-oriented Databases presents

the new definition of the syntax of querying object databases based on the square-bracket syntax used in the collections of objects. The new definition is consistent with the C++ style. Many examples are provided to illustrate both syntax and semantics of the new definition. Implementation of the Querying Operations for Object-oriented Databases will give the implementation schema, interface and test results.

CHAPTER II

LITERATURE REVIEW

Query Languages

Most relational databases have used a single query language -- SQL in all contexts.

SQL is basically the same whether it is embedded in COBOL, PL/I, C, C++, or used in interactively. The database records exist outside the context of any particular application. A relational database can be accessed from programs written in any of a variety of languages using embedded SQL. The syntax and semantics of SQL are inseparable [Loomis, 1995].

The initial specifications for the SQL language dates back to 1973. It was first developed for system R at IBM Research Laboratory, San Jose, California [Chan et al., 1993]. Although not part of relational model theory, SQL is considered by some people to be equal in importance to development of the relational DBMS products [Fleming and Von Hall, 1989]. It was adopted by the American National Standards Institute(ANSI) in 1986 as a standard language for interacting with relational databases. Since the early eighties SQL has prevailed as the database language implemented in most commercial

relational DBMS products. The popularity of SQL and the relational model has prompted the makers of many non-relational database products to provide SQL as a means of access, or “front-end” to their product [Lusardi, 1988]. SQL as a common relational database language enables consistency across product implementations, at least in the way that users, application developers, and, to some extent, database designers interface with the products. Using a common language allows users to deal with only one syntax for invoking those mechanism.

There are other database query languages in addition to SQL. These are done mostly in the laboratories with student users. One famous query language for relational database is QUEL. QUEL is a product of INGRES. It is based on technology published by researchers at the University of California, Berkeley [Loomis, 1995]. QUEL can be used as an interactive query language or it can be embedded within a host programming language. It provides a range of functionality similar to that provided by SQL. But SQL and QUEL were not syntactic variants on a common semantic model. They are totally different. A relational DBMS could not support both. By contrast, an ODBMS can support a variety of surface syntax for the ODMG object-oriented query language semantics.

The Object Database Management Group (ODMG) has defined a query language called the Object Query Language (OQL) [Cattell, 1996]. OQL provides declarative access to objects in the object database. It bases on the ODMG object model and uses the application’s object model as the definition of the database schema. In addition to providing the data access, it also supports the same type defined by the application object model. OQL has the same capabilities as the object-oriented languages, such as object

identity, path expressions to traverse relationships, operation invocation, inheritance, and polymorphism. It relies on the ODMG object model. In 1993 ODMG released his standard: ODMG -- 93 [Cattell, 1996] which includes OQL. This standard OQL provides both a subset and superset of SQL92 [Jordan, 1995]. As a subset, it supports nearly all of the query syntax and semantics of the entry-level ANSI SQL92 specification, which is common subset supported by the relational vendors. As a superset, it provides many extensions to support the object paradigm.

SQL is based on the tuple relational algebra; whereas, OQL is based on domain relational algebra. The two algebra are quite similar, the key difference being that variables in the domain algebra reference values from an arbitrary domain (data type) whereas in the tuple algebra, the variables can reference only tuples.

SQL does not support objects. But the SQL3 committee is preparing a specification of a new SQL language that will support objects. Currently SQL3 only allows objects to serve as the data type of a column, and all object semantics and methods are defined in a procedural extension to SQL. SQL3 must maintain backward compatibility with the huge volume of legacy SQL code deployed in systems today. The current market dominance of SQL is both an asset and a liability of the SQL3 efforts. There are many other fundamental object model and architectural differences between ODMG OQL and SQL3 [Jordan, 1995].

Queries Integrated with the Database Programming Languages

The above querying languages do not share the same type system as ODBMS. They belong to the SQL family. Another family is the integrated family. The members of the family seek seamless integration within the database programming language. A C++

programmer can use C++ to query an object database directly when seamless integration is used. A Smalltalk programmer can use Smalltalk to query an object database.

One approach to object-oriented query syntax is to use standard C++. Some ODBMS provide a Select function that takes a string containing the query as its argument. The SQL statements are coded as strings that C++ ignores. The Select function is inherited by persistent objects whose classes are derived from a virtual base class. The Select function takes as its argument a string form of the query predicate to be applied to the extent of a class. This approach does not use preprocessors and makes debugging programs easier [Loomis, 1994].

Another object query syntax approach is to extend C++ to accommodate query semantics, with careful attention paid to consistency with C++ style so the queries fit seamlessly and naturally with in C++. This approach requires use of a preprocessor, which turns the newly introduced syntax into code that can be handled by the C++ compiler [Loomis, 1994].

Queries can be coded in Smalltalk. An ODBMS can provide a smoothly integrated query capability in Smalltalk using the Select method already defined on Smalltalk collections. This approach implements a seamless interface that is natural to most Smalltalk programmers.

As with Smalltalk, there is another database programming language, OSQL, that combines an expression-oriented procedural language with a high-level, declarative, and optimizable query language [Won Kim, 1995]. The OSQL language combines the object-oriented features found in such as C++ and Smalltalk with a query capability that is a superset of the familiar SQL relational query language. Therefore, OSQL is an object

oriented programming language which provides object identity, a type system with multiple inheritance, polymorphic functions and built-in aggregate object types such as sets and lists. It also provides a declarative query capability similar to that provided by SQL for relational databases [Jordan, 1995].

The Recent Advances in Object-Oriented Databases with Object Query Languages

A new standard (ODMG 2.0) for object-oriented databases was released in 1997 by ODMG [Cattell, 1997]. This completely revised standard for object database management systems represents an important industry consensus on component technology for database products and languages. The goal of the standard is to combine programming languages and database systems into a single environment, providing better performance and more powerful representations for complex database applications. ODMG 2.0 draws upon related work represented by ANSI SQL-92, the OMG Object Model and Interface Definition Language.

There are several enhancements in ODMG 2.0. ODMG has defined a new binding for Sun's Java programming language that combines the safety and pure object orientation of Smalltalk with the more powerful data and syntax of C++, as well as security and automatic program transfer on the internet. The ODMG object model has been modified and expanded to provide a comprehensive specification of object database semantics across many programming languages. A standard for the external form for data and the data schema has been defined to allow data interchange between databases [Wade, 1997]. Some updates and improvements on Object Query Language (OQL) have

been made, too. The new OQL supports method invoking polymorphism during the query process. It can deal with object identity and support operator composition.

Another advance is in SQL3. The current draft includes many new capabilities, such as object functionality and a computationally complete language (PSM). Both of them are very significant additions. PSM is a full language with control flow, procedural operations, and function resolution. It has been defined for both relational and object-oriented databases, including query language, as being compatible with SQL2. The addition of object capabilities includes the ability to define and access Abstract Data Types (ADTs), which have much the same functionality as ODMG objects. PSM is used to implement the internals of objects, both state and operations. The rows in tables may contain objects, which are instances of ADTs. A mechanism to reference objects in other row provides a means to uniquely identify such a row. This is the basis for a concept of identity, similar to ODMG's object reference [Wade, 1997].

Queries in SQL3 include the exact query ability of SQL2, along with an additional ability to invoke methods and traverse relationships. This is much like OQL. OQL queries may apply to and result in objects and collections, whereas SQL3 queries are limited to tables. But, there exist syntax differences between OQL and SQL3, also. For example, OQL uses the dot to indicate traversal, and SQL3 uses double-dot to avoid conflict with the use of dot in SQL2 for naming.

CHAPTER III

OBJECTS AND OBJECT-ORIENTED DATABASES

Object Orientation Concepts

An object is a software building block. An object-oriented technique can be defined loosely as the software modeling and development disciplines that make it easy to construct complex systems from individual components [Khoshafian, 1993]. The objects are implemented through classes. Every object is uniquely identifiable by an object identifier that is unchangeable during the lifetime of the object. A class produces instances, each with the same structure and behavior. A class is composed of two parts: state and operations. State is defined by the values that objects carry for a set of properties. These properties can be either attributes of an object or relationships between the object and one or more other objects. The operations are called methods which manipulate the instances of a class. The state, or data, of an instance is stored in instance variables. The object or class has the following characteristics [Loomis, 1995]:

- **Modularization** Programs are developed in small, understandable modules of data structures and operations allowable on those data.
- **Encapsulation** The code and the data it manipulates are bound together for keeping both safe from outside interference and misuse.

- **Type** All objects that have the same interface, or the same properties, are grouped together. They are treated as being of the same type.
- **Inheritance** The codes are reused through defining new types that have all the properties of another known type, plus some additional properties. The new type inherits the shared properties from the known type.
- **Messaging** An object is made to perform one of its functions by sending to it a message, phrased in a simple, standardized way, independent of how or where the object is implemented.
- **Polymorphism** One interface is allowed to be used for a general class of actions. The specific action is determined by the exact nature of the situation. The system will figure out at runtime exactly which operation code to invoke, based on the type of object that receives the message. Polymorphism is the ability of different kinds of objects to respond to the same message. Message overloading and dynamic binding make polymorphism possible.

Object-oriented Database Concepts

The object-oriented database is the integration of object technology with database capacities. Through object-oriented constructions users can hide the details of implementation of their modules, share objects, and extend their database systems by specializing existing modules. Database users can have the state of objects persist and be updated between various program invocations, and various users can share the same information concurrently. Object-oriented databases can be defined as follows:

Object-oriented databases = Objects + Database capabilities

The capacities of object databases include the following [Khoshafian, 1993]:

- **Persistence** The ability of objects to remain after different program executions. Data manipulated by an object database can be either transient or persistent. Transient data last only for the invocation; they are lost once the program or transaction terminates. Persistent data are retained until they are no longer used. These data are the recoverable objects of the database.
- **Transactions** Units are executed either entirely or not at all. Transactions are atomic. The updates to the persistent database within a transaction either must be visible to the outside world, or none of the updates must be seen.
- **Concurrency Control** The mechanisms limit simultaneous reads and updates by different users, so that all users see a consistent view of data. To guarantee database and transaction consistency, database management systems impose a serializable order of execution.
- **Recovery** The database management systems must guarantee that there is no propagation of the failed partial results or partial updates of transactions in the persistent database.
- **Querying** Queries are used to select subsets and sub-objects from database collections of objects or sets. Queries expressed in terms of high-level languages allow users to qualify what they want to retrieve from the persistent databases.

- **Integrity** Database management systems must keep the database state consistent throughout transactions. The database consistency can be typically expressed through the predicate on the current state of database. Predicates also can apply to objects or attribute values in the database.
- **Security** Database management systems enforce security constraints to access or update persistent objects. The security mechanism, such as granting and revocation of privileges, and the protection of persistent databases from adverse access are integral

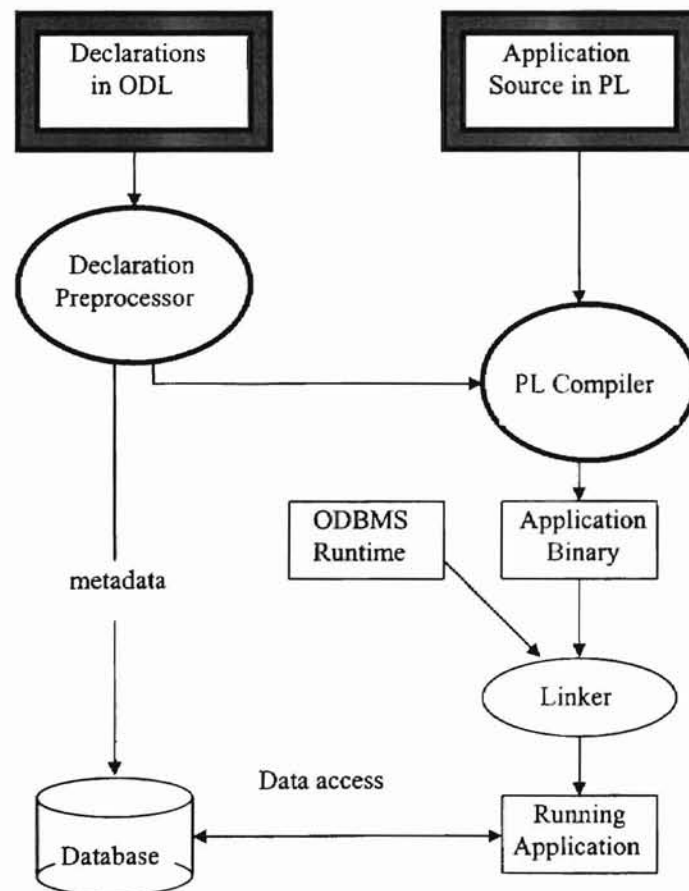


Figure 1 Using an ODBMS

parts of any database management system, including object databases.

Figure 1 illustrates the use of the typical ODBMS product which was introduced by Cattell[Cattell, 1996]. The programmers need to write declarations for the application schema (both data and interface) and a source program for the application implementation. The source program is written in a programming language (PL) such as C++, using a class library that provides full database OML, or Object Manipulation Language, including transactions and object query. Then the declarations and source program are compiled and linked with the ODBMS to produce the running application. The application accesses a new or existing database, whose types must conform to the declarations. Databases may be shared with other application on a network. The ODBMS provides a shared services for transaction and lock management, allowing data to be cached in the application.

An Application Example of Object-oriented Database

Before discussing the method of querying object-oriented database we present an application example of object-oriented database that manages the personnel of a university. This database is called *personnel*. The implementation schema is given in chapter 5 of the thesis.

The university personnel database manages information about students and educational staff (staff, in short). There are five data classes: Person, Student, Staff, TA and Professor. Student and Staff have a common supertype: Person. The type, Person, has properties: name, person_identification, age, address, sex, marriage_status, and an operation: move_address(). An address is a structure whose properties are number,

street, city and zip_code. All instances of **Person** in the database form a collection, *people*. The type, **Student**, inherits **Person**, but has extra properties: major and level, and an operation: assign_major(). The collection of all instances of **Student** in the database is called *students*. The educational staff includes TA and professors. A TA is not only

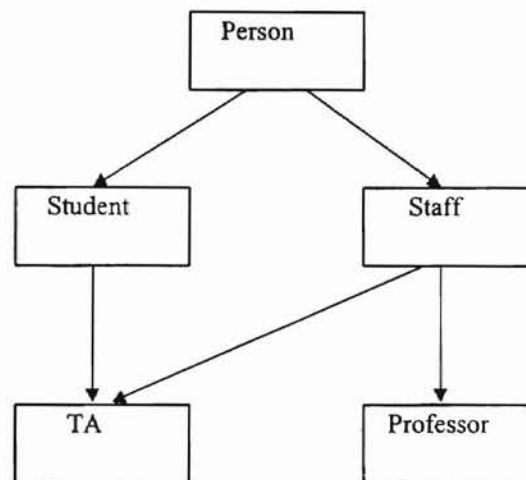


Figure 2 The Data Class Hierarchy in University Personnel Database

student, but also a member of the staff. The type **TA** inherits both of **Student** and **Staff**. All instances of **TA** in the database form a collection, *TAs*. The type **staff** has the extra properties of department and annual_salary and two operations of hire() and fire(). All instances of **Staff** in the database form a collection, *staff*. A professor is a staff and its type **Professor** has the extra property of rank and an operation of grant_tenure(). The collection of all instances of **Professor** in the is called *professors*. Figure 2 is graphical

representation for the application. There is no other relationship among those classes (objects) except inheritance.

CHAPTER IV

A QUERYING METHOD FOR OBJECT-ORIENTED DATABASES

Introduction

The standard ODMG - 93 has given the definition of the Object Query Language (OQL). OQL relies on the ODMG object model and is a superset of the part of standard SQL that deals with database queries. Any SQL sentence which runs on relational tables, works with the same syntax and semantics on collections of ODMG objects. For example, find the ages of persons whose name is Pat from the collection *people*.

```
select distinct x.age  
  
from people x  
  
where x.name="Pat"
```

This statement selects the set of ages, returning a literal of type `set<integer>`. Another example,

```
select distinct struct(a: age, s: sex)  
  
from people x  
  
where x.age>20
```

This does almost the same things, but for each person, it builds a structure containing age and sex. It returns a literal of type `set<struct>`.

Even though this kind of querying language is powerful, it is not natural for a C++ programmer. It is not easy to implement all the syntax and semantics of OQL in a small object-oriented database. There are other problems as described in Chapter 1. The goal is a query facility that is seamlessly integrated with a programming language and easy to use. The result has been extensions to C++ to accommodate query semantics consistent with C++ style. Loomis presented a querying method which applied the square-bracket syntax to the collections of objects [Loomis, 1994]. For the first example above, the query could be phrased as follows:

```
Set<integer> ages = people[name=="Pat"];
```

For the second example, the structure must be defined before using it in a query.

There are other limitations when this approach is used to make complex queries, such as joins. The approach is extended to cover more complex situations. For example, the second query above can be written as follows:

```
Set<struct(integer age, Boolean sex)> age_sex = people[age>20];
```

The term in the square brackets is predicate expression rather than indexes. The *struct* term in the angle brackets is a structure type which has the same meaning as OQL. This kind of querying statement is simple and natural for a C++ programmer to use.

Extension to C++ to Cover the Object Query Language

The above syntax of square brackets on the collection is an extension to C++. In order to meet the complex query requirements we need make more such extensions must be added. Following will give the language definitions for these extensions. A query is an extended C++ statement which consists of a query expression followed by a semicolon. A query expression is built from typed operands using query operators.

Basic Structure

The basic structure of a query expression consists of two operands which separated by the operator =. The first operand defines the query return type which is used for the output of query results. The second operand is the query term which acts on the existing collections with square brackets. The term in the square brackets is the selection predicate.

A typical query statement has the form:

collection type<struct(type var_identifier {, type var_identifier})>

identifier = collection_identifier {, collection_identifier}[selection predicate];

where *var_identifiers* in the *struct* must be properties of the collections, or functions that take the collections as their arguments. *struct* means the all items in the brackets construct a new structure type. All *collection_identifiers* must be the existing collection names. Initially the collections are the extents of the class which generates object data in the object database. *selection predicate* is an Boolean expression or a query condition. For example, find out the faculty's name, annual salary and age at the department 5.

Set<struct(string name, float annual_salary, integer age)> list

```
= staff[department==5];
```

There is several modifiers for the basic structure. One of modifiers is to query a collection directly. It has following form:

```
collection_identifier;
```

For example, the following query statement will list all person's information in the collection, *people*.

```
people;
```

The other modifiers will be described later.

The collection types supported by the ODMG Object Model include

- Set<t>
- Bag<t>
- List<t>
- Array<t>

All the elements in the collection are of the same type t. A Set is an unordered collection of elements with no duplicates allowed. A Bag is an unordered collection of elements that may contain duplicates. A List is an ordered collection of elements. An Array is collection with a fixed number of elements that can be located by position. We can use index number to get the wanted element. For example, to get the third element of an array Names.

```
string third_name = Names[3];
```

Structure Construction

We often need to find more than one property from existing collections. A structure must be built for storing these properties before the query is executed. The definition *struct(type var_identifier {, type var_identifier})* is used to build such a structure whose data members are properties or functions of the collections. When the data types for *var_identifiers* are the same as properties in the collections, then the types of the *var_identifiers* need not to be mentioned. If there is more than one type or collection, we can use a path delimiter. For example, we have a person *p* and want to know the zip code where the person lives.

p.address.zip_code

If a *struct* is known, the *struct* can be omitted. This is another modifier for the basic form. For example, find out all persons whose ages is greater than 30.

Set<Person> spersons = people[age>30];

where Person is the element type of the collection, *people*.

Selection Predicate

The *selection predicate* is an Boolean expression or a query condition which is constructed by relational operator(s) with terms. Here term denotes variable or value and is also an expression. The predicate serves to select only the data matching the predicate. A modifier of *selection predicate* is that it can be empty, with the meaning of a true predicate. For example, select all person's name, age and sex from the collection, *people*.

Set<struct(string name, int age, Boolean sex)> nas = people [];

Aggregate Operators

Aggregate operators are functions that take a collection as input and return a single value. Like SQL we define the 5 aggregate operators {min, max, count, sum, avg}. If e is an expression which denotes a collection, $\langle op \rangle$ is an aggregate operator, then $\langle op \rangle(e)$ is an expression. For example,

`min(collection)`

min operator returns the element with the minimum value in the collection. The max operator returns the element with maximum value in the collection. The count operator returns the number of elements in the collection. The sum operator returns the average value of the elements in the collection. For example, find out all names of persons whose ages are greater than the their average age.

`Set<struct(string name)> names = people[age>avg(people)];`

Relational and Logical Operators

Relational and logical operators in C++ can be used to construct a composite expression in the selection predicate in the square brackets. The relational operations are `==`, `!=`, `<`, `<=`, `>`, and `>=`. Table 1 lists these operators and their meaning. The logical operators are `!`, `&&`, `||`. They are binary operators except `!` which is unary. For example, find the names of all persons whose ages are greater than 50 or less than 35 in department 5.

`Set<struct(string name)> names = people[((age>50) ||
(age<35))&&(department==5)];`

As the same as C++, the two terms before and after any relational operator must have the same types and single values. The expression $e1 <op> e2$ is Boolean expression which can have a value of either True or False only.

Table 1 The Relational Operators

Operator	Meaning
<code>==</code>	equal
<code>!=</code>	not equal
<code><</code>	less than
<code><=</code>	less than or equal
<code>></code>	greater than
<code>>=</code>	greater than or equal

in and !in Operators

If $e1$ and $e2$ are expressions, $e2$ is a collection, and $e1$ has the type of elements of $e2$, then $e1 \text{ in } e2$ and $e1 \text{ !in } e2$ are expressions. Here `!` means not. If elements $e1$ belongs to collection $e2$, then $e1 \text{ in } e2$ returns true, $e1 \text{ !in } e2$ returns false, otherwise, $e1 \text{ in } e2$ returns false, $e1 \text{ !in } e2$ returns true. For example, to find the students whose names are in the list, `lis`.

```
Set<struct(string name, unsigned long person_id, String major)> nim  
  
= students[ name in lis];
```

Nested Queries

The query method supports nested queries by applying a query to the collections which are the results of the preceding queries. For example, assuming we know the collections: staff and students, find out the names and majors of students who are TAs of department 1.

```
Set<struct(string name)> staffnames = staff [department==1];  
  
Set<struct(string name, string major)> TAnames =  
  
students [(name in facultynames) && (level==PhD)];
```

The nested queries cannot be in the select predicate since each collection must be declared early before getting in the execution. All collections in the select predicate expression must exist, otherwise raise an exception.

Join Query

As with SQL, the query method allows computation of joins among many collections. For example, select the people who bear the name of a flower, assuming there exists a set of all flowers called Flowers.

```
Set<Person> person1 = people[ name == Flowers.name];
```

Duplicates and Order

The collection has 4 different types: Set, Bag, List, and Array. The Set collection does not allow the duplicate elements. This corresponds to the key word *distinct* in the select clause of SQL. But the Bag collection allows duplicate elements. For example,

find out the names of people whose ages are 30.

```
Bag<struct(string name)> names = people[ age==30 ];
```

The Bag names has duplicate elements since there could be several persons who have the same ages and the same names.

Sometimes an ordered collection is needed. This time the collection list can be applied. For example, determine the students whose majors are computer science ordered by age.

```
List<struct(string name, integer age, integer level)> student1 =  
students[major == "computer science"];
```

The Object Query Language Grammar

The above section has discussed the definition of the object query language. Following is the grammar of the object query language.

```
Query → collection <expression1> id = id [ expression2 ]; | id;
```

```
collection → Set | Bag | List | Varray
```

```
expression1 → type | struct ( term1 )
```

```
term1 → term1, term1 | type id
```

```
type → int | float | char | double | long | string
```

```
expression2 → (expression2) logop (expression2) | term2 relop term2 | space
```

```
term2 → id | const | aggregator (id)
```

logop \rightarrow **&&** | **||**

relop \rightarrow **=** | **!=** | **>** | **>=** | **<** | **<=** | **in** | **!in**

const \rightarrow **Number** | **String**

aggregator \rightarrow **min** | **max** | **avg** | **count** | **sum**

where

- collection denotes which kind of collection.
- expression1 denotes type or structure.
- expression2 denotes selection predicate.
- term1 denotes a variable and its type.
- term2 denotes identifier, value or aggregator function.
- logop denotes logical operator.
- relop denotes relational operator.
- const denotes number or string.
- bold symbols denote they are terminated symbols

CHAPTER V

IMPLEMENTATION OF THE QUERYING OPERATION FOR OBJECT-ORIENTED DATABASES

Implementation Schema of An Object-oriented Database

This section presents the implementation schema definitions of the university personnel management database. The university personnel management database has five data classes: Person, Student, Staff, TA and Professor. Figure 3.2 shows the relationship among these classes. Each of these classes has specific data members and methods which are discussed in the section entitled Object and Object-oriented Database. All Data are stored in the objects which are instances of data classes. At this point it is appropriate to present the E-R model of the university personnel management database from the view of the relational database.

Figure 3 is the E-R schema of the university personnel management database. But unlike the object model, the E-R model cannot have many inheritances. Therefore a TA inherits either a Student or a Staff only. Figure 3 supposes a TA inherits a Student. Figure 4 is the tabular representation of the E-R schema. There is much redundancy in it

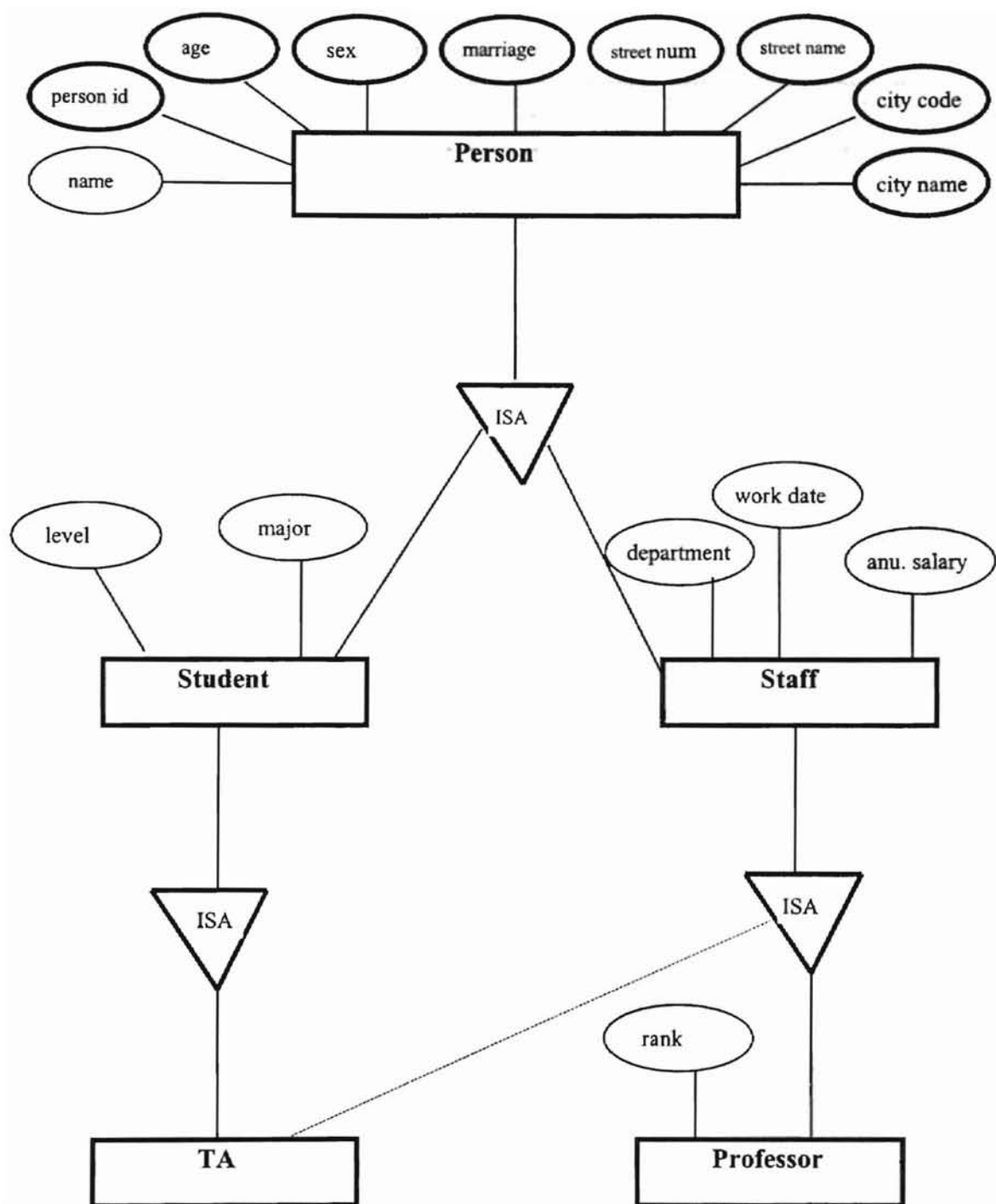


Figure 3 E-R Schema

Person

name	person id	age	sex	marriage	street num.	street name	city code	city name
------	-----------	-----	-----	----------	-------------	-------------	-----------	-----------

Student

name	person id	level	major
------	-----------	-------	-------

Staff

name	person id	work date	ann. salary	department
------	-----------	-----------	-------------	------------

TA

name	person id	level	major	work date	ann. salary	department
------	-----------	-------	-------	-----------	-------------	------------

Professor

name	person id	work date	department	ann. salary	rank
------	-----------	-----------	------------	-------------	------

Figure 4 The Tabular Representation

since the E-R model simply cannot represent the complicated relationship of inheritances. However using the object model this kind relationship can be easily represented as inheritance without any redundancy. This is one of the advantages that the object-oriented database has. The university personnel database is the object-oriented database which follow the standard: ODMG-93 [Cattell 93] except the querying method.

Object Class and Data Classes

As mentioned before there are five data classes in this database. They are Person, Student, Staff, TA and Professor. These classes inherit the Object class. The Object class is introduced to allow the type definer to specify when a class is capable of having persistent as well as transient instances. Instances of classes derived from Object can be

```
class d_Object {
public:
    d_Object();
    d_Object(const d_Object & ob);
    virtual ~d_Object();
    d_Object & operator=(const d_Object & ob);
    void mark_modified();
    void * operator new(size_t size);
    void * operator new(size_t size, const d_Ref_Any &cluster, const char
        *typename);
    void * operator new(size_t size, d_Database *database, const char
        *typename=" ");
    void operator delete(void * ptr);
    virtual void saveData(ofstream &out) const;
    virtual void restoreData(istream &in) ;
    virtual void printData();
private:
    int markbit;
};
```

Figure 5 Schema Definition of the Class d_Object

either persistent or transient. Figure 5 is the schema definition of the Object class (here putting prefix d_ before Object means it is used in the database).

Figure 6 - 10 define the schema of class Person, Student, Staff, TA, and Professor respectively. They all inherit class d_Object.

```
class Person: public d_Object {
public:
    char name[21];
    long personid;
    int age;
    int streetnum;
    char streetname[51];
    long citycode;
    char cityname[51];
    int sex;
    int marriage;
    Person();
    Person(const char *pname, long pid, const char *bd, int sx, int mar);
    void move(const Address &);

    //people is collection of instances of Person
    static d_Ref<d_Set<Person> people;
    static const char * const extent_name ;
    virtual void saveData(ofstream &out) const;
    virtual void restoreData(istream &in);
    virtual void printData();
};
```

Figure 6 Schema Definition of the Class Person

```
class Student: public Person {
public:
    char major[51];
    int level;
    Student();
    Student(int lv);
    Student(const char *pname, long pid, const char *bd, int sx, int mar, int
        lv);
```

```

~Student();
void assign_major(const char *mj);

//students is collection of instances of Student
static d_Ref<d_Set<Student>> students;
static const char * const extent_name ;
virtual void saveData(ofstream &out) const;
virtual void restoreData(istream &in);
virtual void printData();
};

```

Figure 7 Schema Definition of the Class Student

```

class Staff: public Person {
public:
    float an_salary;
    int department;
    char workdate[51];
    Staff();
    Staff(float ans, int dep);
    Staff(const char *pname, long pid, const char *bd, int sx, int mar, float
        ans, int dep);
    ~Staff();
    void hire(const char *hda);

    //staff is collection of instances of Staff
    static d_Ref<d_Set<Staff>> staff;
    static const char * const extent_name ;
    virtual void saveData(ofstream &out) const;
    virtual void restoreData(istream &in);
    virtual void printData();
};

```

Figure 8 Schema Definition of the Class Staff

```

class TA: public Student {
public:
    float an_salary;
    int department;
    char workdate[51];

```

```

    TA();
    TA(const char *pname, long pid, const char *bd, int sx, int mar,int lv,
        float ans, int dep, char *wd);
    ~TA();
    //TAs is collection of instances of TA
    static d_Ref<d_Set<TA>> TAs;
    static const char * const extent_name ;
    virtual void saveData(ofstream &out) const;
    virtual void restoreData(istream &in);
    virtual void printData();
};

```

Figure 9 Schema Definition of the Class TA

```

class Professor: public Staff {
public:
    int rank;

    Professor();
    Professor(const char *pname, long pid, const char *bd, int sx, int mar,
        float ans, int dep);
    ~Professor();

    //professors is collection of instances of Professor
    void grant_tenure(int rk);
    static d_Ref<d_Set<Professor>> professors;
    static const char * const extent_name ;
    virtual void saveData(ofstream &out) const;
    virtual void restoreData(istream &in);
    virtual void printData();
};

```

Figure 10 Schema Definition of the Class Professor

In the above definitions there are some new types, such as `d_Ref_Any`, `d_Ref<T>`, `d_Set<T>`, and `d_Database`, which will be described later.

Reference Classes

Sometimes some objects need to refer to other objects. It can be achieved through a smart pointer or reference called a `d_Ref`. A `d_Ref<T>` is a template class

```
template <class T> class d_Ref {
private:
    T * ptr;
public:
    d_Ref();
    d_Ref(T *fromPtr);
    d_Ref(const d_Ref<T> &ref);
    d_Ref(const d_Ref_Any &any);
    ~d_Ref();
    operator d_Ref_Any() const;
    d_Ref<T> & operator =(T *fromPtr);
    d_Ref<T> & operator =(const d_Ref<T> &ref);

    void clear();
    T * operator ->() const;
    T & operator *() const;

    T * Ptr() const
    void delete_object();
    operator const void *() const;
    int operator !() const;
    int is_null() const;
    friend int operator ==(const d_Ref<T> &refL, const d_Ref<T> &refR);
    friend int operator ==(const d_Ref<T> &refL, const T *ptrR);
    friend int operator ==(const T *ptrL, const d_Ref<T> &refR);
    friend int operator ==(const d_Ref<T> &L, const d_Ref_Any &R);
    friend int operator ==(const d_Ref_Any &L, const d_Ref<T> &R);
    friend int operator !=(const d_Ref<T> &refL, const d_Ref<T> &refR);
    friend int operator !=(const d_Ref<T> &refL, const T *ptrR);
    friend int operator !=(const T *ptrL, const d_Ref<T> &refR);
    friend int operator !=(const d_Ref<T> &L, const d_Ref_Any &R);
    friend int operator !=(const d_Ref_Any &L, const d_Ref<T> &R);
};
```

Figure 11 Schema Definition of the Template Class `d_Ref`

which refers to an instance of type T. There is also a `d_Ref_Any` class defined to support any type. Figure 11 is the schema definition of `d_Ref`.

The `d_Ref_any` class is defined that provides a generic reference to any type. Its main purpose is to handle generic references and allow conversions of `d_Refs` between the different types. A `d_Ref<T>` can always be converted to a `d_Ref_Any` through a function in the `d_Ref<T>` template class. Using the constructor and assignment operator in the template class a `d_Ref_Any` can become a `d_Ref`. The `d_Ref_any` is defined in Figure 12.

```
class d_Ref_Any {
public:
    d_Object * ptr;
    d_Ref_Any();
    d_Ref_Any(const d_Ref_Any &any);
    d_Ref_Any(d_Object *obj);
    ~d_Ref_Any();
    d_Ref_Any & operator =(const d_Ref_Any &any);
    d_Ref_Any & operator =(d_Object *obj);
    void clear();
    void delete_object();
    operator const void *() const;
    int operator !() const;
    int is_null() const;
    friend int operator ==(d_Ref_Any &refL, d_Ref_Any &refR);
    friend int operator ==(d_Ref_Any &refL, d_Object *obj);
    friend int operator ==(d_Object *obj, d_Ref_Any &refR);
    friend int operator !=(d_Ref_Any &refL, d_Ref_Any &refR);
    friend int operator !=(d_Ref_Any &refL, d_Object *obj);
    friend int operator !=(d_Object *obj, d_Ref_Any &refR);
};
```

Figure 12 Schema Definition of the Class `d_Ref_Any`

Collection Template Classes

Collection template classes are used to represent a collection whose elements are of any type. A conforming implementation must support at least the following subtypes of `d_Collection`: `d_Set`, `d_Bag`, `d_List` and `d_Varray`. Class `d_Collection` is an abstract class which cannot have instances. It is derived from object class `d_Object`, allowing instances of concrete classes derived from `d_Collection` to be stand-alone persistent objects. There are two classes, `Tstruct<T>` and `d_Iterator<T>`, involved in the class `d_Collection`. The template class `Tstruct` is used to construct the linked nodes in the collection. Figure 13 shows its schema definition. The template class `d_Iterator` defines the generic behavior for iteration. Its purpose is for sequentially returning each element from the collection over which the iteration is defined. Normally, an iterator is initialized by the `create_iterator` method on the collection class. Figure 14 lists the schema definition for `d_Iterator`.

```
Template <class T> class Tstruct {
public:
    T tptr;
    Tstruct<T> *next;
    Tstruct();
};
```

Figure 13 Schema Definition of the Template Class `Tstruct`

```
template <class T> class d_Iterator {
public:
    Tstruct<T> *Inode, *Onode;
    d_Iterator();
    d_Iterator(const d_Iterator<T> &it);
```

```

~d_iterator();
d_iterator<T> & operator =(const d_iterator<T> &it);
friend int operator ==(const d_iterator<T> &itL, const d_iterator<T>
                        &itR);
friend int operator !=(const d_iterator<T> &itL, const d_iterator<T>
                        &itR);
void reset();
int not_done() const;
void advance();
d_iterator<T> & operator ++();
d_iterator<T> & operator --();
T get_element() const;
T operator *() const;
void replace_element(const T &elem);
int next(T &obj);
};

```

Figure 14 Schema Definition of the Class d_iterator

Figure 15 lists the schema definition of the template class d_Collection.

```

template <class T> class d_Collection: public d_Object {
protected:
    Tstruct<T> *Head, *Cnode;
    unsigned long size;
    int order, duplicate;

    d_Collection():d_Object();
    d_Collection(const d_Collection<T> &dc);
    d_Collection<T> & operator =(const d_Collection<T> &dc);

public:
    virtual ~d_Collection();
    d_Collection<T> & assign_from(const d_Collection<T> &dc);
    friend int operator ==(const d_Collection<T> &cL, const
                          d_Collection<T> &cR);
    friend int operator !=(const d_Collection<T> &cL, const d_Collection<T>
                          &cR);
    unsigned long cardinality() const;
    int is_empty() const;
    int is_ordered() const;
};

```



```

int allows_duplicates() const;
int contains_element(const T &element) const;
void insert_element(const T &elem);
void remove_element(const T &elem);
void remove_all();
d_iterator<T> create_iterator() const;
d_iterator<T> begin() const;
d_iterator<T> end() const;
// query
const T & select_element(const char *OQL_predicate) const;
d_iterator<T> select(const char *OQL_predicate) const;
int query(d_collection<T> &, char *OQL_predicate) const;
int exists_element(char *OQL_predicate) const;

};

```

Figure 15 Schema Definition of the Class d_collection

The class d_Set<T> is an unordered collection of elements of type T with no duplicates. Figure 16 shows the schema definition of d_Set.

```

template <class T> class d_Set: public d_collection<T> {
public:
    d_Set();
    d_Set(const d_Set<T> &set);
    ~d_Set();
    d_Set<T> & operator =(const d_Set<T> &set);
    d_Set<T> & union_of(const d_Set<T> &sL, const d_Set<T> &sR);
    d_Set<T> & union_with(const d_Set<T> &s2);
    d_Set<T> operator +=(const d_Set<T> &s2);
    d_Set create_union(const d_Set<T> &s) const;
    friend d_Set<T> operator +(const d_Set<T> &s1, const d_Set<T> &s2);
    d_Set<T> & intersection_of(const d_Set<T> &sL, const d_Set<T> &sR);
    d_Set<T> & intersection_with(const d_Set<T> &s2);
    d_Set<T> & operator *=(const d_Set<T> &s2);
    d_Set<T> create_intersection(const d_Set<T> &s) const;
    friend d_Set<T> operator *(const d_Set<T> &s1, const d_Set<T> &s2);
    d_Set<T> & difference_of(const d_Set<T> &sL, const d_Set<T> &sR);
    d_Set<T> & difference_with(const d_Set<T> &s2);

```

```

    d_Set<T> & operator -=(const d_Set<T> &s2);
    d_Set<T> create_difference(const d_Set<T> &s) const;
    friend d_Set<T> operator -(const d_Set<T> &s1, const d_Set<T> &s2);
};

```

Figure 16 Schema Definition of the Template Class d_Set

The class d_Bag<T> is an unordered collection of elements of type T that does allow duplicate values. Figure 17 is the schema definition of class d_Bag.

```

template <class T> class d_Bag: public d_Collection<T> {
public:
    d_Bag;
    d_Bag(const d_Bag<T> &bag);
    ~d_Bag();
    d_Bag<T> & operator =(const d_Bag<T> &bag);
    d_Bag<T> & union_of(const d_Bag<T> &sL, const d_Bag<T> &sR);
    d_Bag<T> & union_with(const d_Bag<T> &s2);
    d_Bag<T> operator +=(const d_Bag<T> &s2);
    d_Bag<T> create_union(const d_Bag<T> &s) const;
    friend d_Bag<T> operator +(const d_Bag<T> &s1, const d_Bag<T> &s2);
    d_Bag<T> & intersection_of(const d_Bag<T> &sL, const d_Bag<T> &sR);
    d_Bag<T> & intersection_with(const d_Bag<T> &s2);
    d_Bag<T> & operator *=(const d_Bag<T> &s2);
    d_Bag<T> create_intersection(const d_Bag<T> &s) const;
    friend d_Bag<T> operator *(const d_Bag<T> &s1, const d_Bag<T> &s2);
    d_Bag<T> & difference_of(const d_Bag<T> &sL, const d_Bag<T> &sR);
    d_Bag<T> & difference_with(const d_Bag<T> &s2);
    d_Bag<T> & operator -=(const d_Bag<T> &s2);
    d_Bag<T> create_difference(const d_Bag<T> &s) const;
    friend d_Bag<T> operator -(const d_Bag<T> &s1, const d_Bag<T> &s2);
};

```

Figure 17 Schema Definition of the Template Class d_Bag

The template class `d_List` is an ordered collection of elements of type `T` and does allow for duplicate values. The beginning index value of the `d_List` is 0, following the convention of C and C++. The figure 18 shows the schema definition of the template class `d_List`.

```
template <class T> class d_List: public d_Collection<T> {
public:
    d_List(const d_List<T> & list): d_Collection<T>(list);
    ~d_List();
    d_List<T> & operator =(const d_List<T> & list);
    const T & retrieve_first_element() const;
    const T & retrieve_last_element() const;
    void remove_first_element();
    void remove_last_element();
    const T & operator [] (unsigned long position);
    int find_element(const T &element, unsigned long &position) const;
    const T & retrieve_element_at(unsigned long position) const;
    void remove_element_at(unsigned long position);
    void replace_element_at(const T &element, unsigned long position);
    void insert_element_first(const T &element);
    void insert_element_last(const T &element);
    void insert_element_before(const T &element, unsigned long position);
    void insert_element_after(const T &element, unsigned long position);
    d_List<T> & concat(const d_List<T> &listR) const;
    friend d_List<T> operator +(const d_List<T> &listL, const d_List<T>
        &listR);
    d_List<T> & append(const d_List<T> &listR);
    d_List<T> & operator +=(const d_List<T> &listR);
};
```

Figure 18 Schema Definition of the Template Class `d_List`

The template class `d_Varray` is a one-dimension array of varying length consisting of type `T`. In the same manner as the `d_List` the beginning index value of the `d_Varray` is 0. Figure 19 lists the `d_Varray` schema definition.

```

// template class d_Varray
template <class T> class d_Varray: public d_Collection<T> {
private:
    unsigned long maxsize;
public:
    d_Varray();
    d_Varray(unsigned long length);
    d_Varray(const d_Varray<T> &dv);
    ~d_Varray();
    d_Varray<T> & operator =(const d_Varray<T> &dv);
    unsigned long upper_bound() const;
    void resize(unsigned long length);
    const T & operator [](unsigned long index);
    int find_element(const T &element, unsigned
    const & retrieve_element_at(unsigned long index) const;
    void remove_element_at(unsigned long index);
    void replace_element_at(const T &element, unsigned long index);
};

```

Figure 19 Schema Definition of the Template Class d_Varray

Database Structures and System Classes

All objects of data classes are stored in the database. The data structure used to store objects is listed in Figure 20.

```

struct objcollstruct {
    int size;
    char objectname[maxsize1][51];
    d_Object *objectcoll[maxsize1];
};

struct objectstruct {
    int typesize, objsize[maxsize1];
    d_Object *objectda[maxsize1][maxsize2];
    char typenames[maxsize1][51];
};

```

```

struct DBstruct {
    struct objcollstruct *objcoll;
    struct objectstruct *objects;
    struct objectstruct *dbobjs;
    struct objectstruct *delobjs;
};

typedef struct DBstruct *dbase;

```

Figure 20 Schema Definition of Data Store Structure

An ODBMS provides a type `d_Transaction` which has `begin`, `commit`, `abort` and `checkpoint` methods. The `begin` method starts a transaction. Transactions must be explicitly created and started. The `commit` method commits all persistent objects created, deleted, or modified within the transaction to the database and releases any locks held by the transaction. The `abort` method aborts all changes to objects and releases the locks. The `checkpoint` method commits objects modified within the transaction since the last checkpoint, but maintain all locks it held on those objects. Figure 21 is class `d_Transaction` schema definition.

```

class d_Transaction {
public:
    d_Transaction();
    ~d_Transaction();
    void begin();
    void commit();
    void abort();
    void checkpoint();
private:
    d_Transaction(const d_Transaction &);
    d_Transaction & operator=(const d_Transaction &);
};

```

Figure 21 Schema Definition of the Class `d_Transaction`

There is a predefined type `d_Database` supplied by the ODBMS. An ODBMS may manage one or more logical databases which are instances of the class `d_Database`. The class `d_Database` must have open and close methods. The open method must be invoked, with a database name as its argument, before any access can be made to the persistent objects in the database. The close method must be invoked when a program has finished all access to the database. Figure 22 shows the database class schema definition.

```
class d_Database {
public:
    d_Database();
    ~d_Database();
    dbase RecentDB;
    char RecentDBname[51];
    int trans;
    static const d_Database * const transient_memory=NULL;
    enum access_status {not_open, read_write, read_only, exclusive};
    void open(const char * database_name); //, access_status
        status=read_write);
    void close();
    void set_object_name(const d_Ref_Any &theObject, const char
        *newName);
    void rename_Object(const char *oldName, const char *newName);
    d_Ref_Any Lookup_Object(const char *name) const;

private:
    d_Database(const d_Database &);
    d_Database & operator =(const d_Database &);
};
```

Figure 22 Schema Definition of Class `d_Database`

Design and Implementation of the Query Method

Chapter 4 described the new object query language that is an extension to C++.

In this section the semantics of the new object query language are mapped into the C++

language. A query class, three symbol tables and a query execution function are constructed to execute the new query method. Figure 23 lists the implementation schema of the three symbol tables included collTable, claTable and symTable. The CollTable is provided to store the collection address, name and element type name. The ClaTable stores the information of element types in all collections, such as type name, its data member names and types. The SymTable is used to store the token symbols during the parsing time. Here a class string is used as type for all names and defined in Figure 24.

```

struct clatab1 {
    string classname;
    int memsize;
    int memtype[20];
    string memname[20];
};

struct clatab2 {
    int tablesize;
    struct clatab1 table[maxsize];
};

struct clatab2 ClaTable;

struct colltab1 {
    string collname;
    string classname;
    d_Object *objaddr;
};

struct colltab2 {
    int tablesize;
    struct colltab1 table[maxsize];
};

struct colltab2 CollTable;

```

```

struct symtab1 {
    string symname;
    int token;
};

struct symtab2 {
    int tablesize;
    struct symtab1 table[maxsize];
};

struct symtab2 SymTable;

```

Figure 23 Three Symbol Tables

```

class string {
private:
    char *str;
public:
    string();
    string(char *s);
    string(string& s);
    string& operator=(string& a);
    string& operator=(char *s);
    string& operator=(const int a);
    operator const char *() const;
    char & operator[](int index);
    int length() const;
    void print() {cout<<str<<endl;}
    ~string();

    friend int operator==(const string &sl, const string &sr);
    friend int operator==(const string &sl, const char *pr);
    friend int operator==(const char *pl, const string &sr);
    friend int operator!=(const string &sl, const string &sr);
    friend int operator!=(const string &sl, const char *pr);
    friend int operator!=(const char *pl, const string &sr);
    friend int operator<(const string &sl, const string &sr);
    friend int operator<(const string &sl, const char *pr);
    friend int operator<(const char *pl, const string &sr);
    friend int operator<=(const string &sl, const string &sr);
    friend int operator<=(const string &sl, const char *pr);
    friend int operator<=(const char *pl, const string &sr);

```



```

        friend int operator>(const string &sl, const string &sr);
        friend int operator>(const string &sl, const char *pr);
        friend int operator>(const char *pl, const string &sr);
        friend int operator>=(const string &sl, const string &sr);
        friend int operator>=(const string &sl, const char *pr);
        friend int operator>=(const char *pl, const string &sr);
};

```

Figure 24 Schema Definition of the Class string

The query class, queryclass, is a general type used for the query results. All data members of queryclass have the same type, string. Figure 25 shows queryclass schema definition.

```

class queryclass {
public:
    string *ab;
    int size;

    queryclass(int sz=20);
    ~queryclass();
    string & operator[](int i);
    void print_query();

};

typedef d_Ref<queryclass> querycla;

```

Figure 25 Schema Definition of the Class queryclass

The query execution function is implemented through a class exequery. Figure 26 is the implementation schema definition of the class exequery. There are two types involved in the class exequery. One is the class stack. Another is the structure

termstruct. Both of them are used to calculate the value of selection predicate. They are presented at the top of figure 26. The query execution function is listed in figure 27. The query execution function takes a string containing the query statement as its argument. This string then is passed to the constructor of class exequery. The methods, collparser and predparser, parse and evaluate the query. If there is not any error, the query results are outputted through the method qprint. If there exists any error in the query statement, the function will stop and print out the sentence: "syntax error".

```
// class stack definition
class stack {
private:
    int size;
    int array[100];
public:
    stack();
    ~stack();
    int isfull();
    int isempty();
    void push(int elem);
    int pop();
    int top();
};

// structure termstruct used to store both side terms of predicate
struct termstruct {
    string coll;
    string mem;
    int memtype;
};

// class exequery definition
class exequery {
private:
    char currstr[81];
    int error;
    char collstr[81];
    char selpred[81];
    int pos;
```

```

char tokenstr[81];
int lookahead;
int colltypenum;
int memnum;
int newclass;
struct colltab1 currcoll;
struct colltab1 qcoll;
struct clatab1 currclass;
struct clatab1 qclass;
stack stk;
struct termstruct term[2];
int termnum;
int relnum;
int lognum;
int termvalue, single;
void initsymTable();
int lexan();
void stmt();
void expr1();
void expr2(querycla &elem);
void colltype();
void term1();
void term2(querycla &elem);
void calstack(querycla &elem);
void calvalue(querycla &elem);
void calagg(int i, int j, char *str);
void savequery();
void setqelem(d_Collection<querycla> *qcol, querycla &elem);
int looksym(string str);
int lookclass(string tname);
int lookcoll(string collname);
void setqclass(int j);
void setnewcoll();
void setoldcoll();
void match(int num);
public:
    exequery(char * qs);
    exequery(string qs);
    ~exequery();
    void collparser();
    void predparser();
    void qprint();
};

```

Figure 26 Schema Definition of the Class Exequery

```

Void Query_execute(string query) {
    exequery q(query);
    q.collparser();
    q.predparser();

    q.qprint();
}

```

Figure 27 Implementation Definition of Query Execution Function

Test Results

The preceding two sections define the entire schema of the university personnel management database. The details of the implementation are not be presented since they are very complicated and much longer. This section shows only the execution results of the university personnel management database. When running this program, a main menu is shown. Figure 28 is its main menu. There are 9 choices. The program prompts the user to enter a choice number. Number 1 opens a database. A database name must be entered. The university personnel management database name is personnel. A database must be opened before any other choices are made except number 9, Exit. Then to modify the objects in the database, start a transaction, or choose number 2. Number 3 and 4 are provided to add and delete the objects in the database. When choosing number 3 or 4, there a sub-menu is presented. Figure 29 is the sub menu which provides five data class choices and one return. One must choose one data class to create objects and enter the associated data. When modifying objects in the database is completed, one must commit the transaction if the updates are to be retained permanently or abort the

transaction if undesired. Once the transaction is committed it cannot be aborted. All modifications since the last transaction are permanently kept in the database except modifying again. If choosing number 7, the current database can be queried. The query topic will be discussed in detail later. When all operations are completed on the current database it can be closed, or choose number 8. At this moment the program can be exited (choose number 9) or opened to a database again.

```

*****
*           1. Open a Database           *
*           2. Start Transactions         *
*           3. Add Objects                *
*           4. Delete Objects             *
*           5. Commit Transactions        *
*           6. Abort Transactions         *
*           7. Query Current Database    *
*           8. Close Current Database     *
*           9. Exit                      *
*****
Please make a choice:

```

Figure 28 The Main Menu

The current database can be queried by choosing number 7 after a database is opened. The program prompts the user to enter a query statement. When a query statement is entered, the program compiles the query statement, then executes the query on the database at run-time. If no errors occur, the query result is output; otherwise the phrase "Syntax Error" is printed. For example, in response to the query "people;" the program lists all persons' information in the university personnel management database as shown in Table 2. Tables 3 - 6 present the information for the four other data classes.

```

*****
*           1. Person           *
*           2. Student          *
*           3. Staff            *
*           4. TA               *
*           5. Professor        *
*           6. Return           *
*****
Please select a class number:

```

Figure 29 The Sub Menu

When the program finishes a query, it asks the question: “Do you need more queries (y/n)?”. If the user responds “yes” the database can be queried continually; otherwise, a “no” response cause a return to the main menu.

If the following query statement is entered,

```

d_Set<struct(string name, long personid, int age)> listname =people[
    cityname=="Stillwater" ];

```

the program presents a list of names, identities and ages of persons who live in Stillwater.

The query results are presented in Table 7.

The program supports the nested queries by applying a query to the collections that are the query results of the preceding queries. For example, to determine the names and majors of students who are TA of department 1, the following two query statements can be used:

```

d_Set<struct(string name)> staffnames = staff [ department ==1];
d_Set<struct(string name, string major)> TAnames = students[ name in
    staffnames];

```

Table 8 shows the query results. However, the same results can be elicited with the following query statement:

```
d_Set<struct( string name, string major )> TAnames = TAs[department==1];
```

Table 2 The Query Results of the Collection, people

name	personid	age	sex	marriage	streetnum	streetname	citycode	cityname
Tang	462993398	35	1	1	44	University	74075	Stillwater
Wang	336445880	32	0	1	3020	Western	74075	Stillwater
Li	564380010	29	0	1	12	University	74075	Stillwater
Johnson	430710038	24	1	0	3041	1 st	74104	Tulsa
Wu	398441042	25	0	0	402	81ave.	74106	Tulsa
Stone	333128899	45	1	1	3404	Vermont	45056	OKC
Brown	345224677	34	1	1	568	Washington	74075	Stillwater
Hua	456224456	30	0	1	4506	Oregon	45078	OKC
Lee	456333123	50	0	0	3565	4ave.	74075	Stillwater
William	234676778	42	1	1	450	Lewis	5124	Tulsa

Table 3 The Query Results of the Collection, students

name	personid	age	sex	marriage	streetnum	streetname	citycode	cityname	level	major
Tang	462993398	35	1	1	44	University	74075	Stillwater	5	computer
Wang	336445880	32	0	1	3020	Western	74075	Stillwater	4	physics
Li	564380010	29	0	1	12	University	74075	Stillwater	5	computer
Johnson	430710038	24	1	0	3041	1 st	74104	Tulsa	3	chemistry
Wu	398441042	25	0	0	402	81ave.	74106	Tulsa	6	computer
Hua	456224456	30	0	1	4506	Oregon	45078	OKC	4	computer
William	234676778	42	1	1	450	Lewis	75124	Tulsa	3	civil

Table 4 The Query Results of the Collection, staff

name	personid	age	sex	marriage	streetnum	streetname	citycode	cityname	workdate	ann_salary	department
Tang	462993398	35	1	1	44	University	74075	Stillwater	01/01/98	1000.0	1
Li	564380010	29	0	1	12	University	74075	Stillwater	08/16/97	800.0	1
Johnson	430710038	24	1	0	3041	1 st	74104	Tulsa	01/13/97	850.0	3
Stone	333128899	45	1	1	3404	Vermont	45056	OKC	01/10/90	60000.0	1
Brown	345224677	56	1	1	568	Washington	74075	Stillwater	08/16/95	45000.0	1
Lee	456333123	50	0	0	3565	4ave.	74075	Stillwater	01/14/80	85000.0	2
William	234676778	42	1	1	450	Lewis	75124	Tulsa	06/01/97	1000.0	4

Table 5 The Query Results of the Collection, TAs

name	personid	age	sex	marriage	streetnum	streetname	citycode	cityname	level	major	workdate	ann_salary	department
Tang	462993398	35	1	1	44	University	74075	Stillwater	5	computer	01/01/98	1000.0	1
Li	564380010	29	0	1	12	University	74075	Stillwater	5	computer	08/16/97	800.0	1
Johnson	430710038	24	1	0	3041	1 st	74104	Tulsa	3	chemistry	01/13/97	850.0	3
William	234676778	42	1	1	450	Lewis	75124	Tulsa	3	civil	06/01/97	1000.0	4

Table 6 The Query Results of the Collection, professors

name	personid	age	sex	marriage	streetnum	streetname	citycode	cityname	workdate	annual_salary	department	rank
Stone	333128899	45	1	1	3404	Vermont	45056	OKC	01/10/90	60000.0	1	2
Brown	345224677	56	1	1	568	Washington	74075	Stillwater	08/16/95	45000.0	1	1
Lee	456333123	50	0	0	3565	4ave.	74075	Stillwater	01/14/80	85000.0	2	3

Table 7 The Query Results of “People Who Live in Stillwater”

name	personid	age
Tang	462993398	35
Wang	336445880	32
Li	564380010	29
Brown	345224677	34
Lee	456333123	50

Table 8 The Query Results of “TA in Department 1”

name	major
Tang	computer
Li	computer

The aggregate operators can be tested on the personnel database. For example, the following query statements can be used to find out all the names and ranks of professors whose ages are greater than the average age in department 1.

```
d_Set<struct( int age)> ages = staff[ department == 1];
```

```
d_Set<struct( string name, int rank)> proflist = professors [ (age>avg( ages))
```

```
&&(department ==1)];
```

Table 9 is the final query results.

Table 9 The Query Results of “Professors Whose Ages Are Greater Than the Average Age in Department 1”

name	rank
stone	2

The Table 2 - 9 list some typical query results in the university personnel database. Of course, more examples can be made.

CHAPTER VI

SUMMARY, CONCLUSION AND SUGGESTED FUTURE WORK

The concepts of object orientation have been accepted well [Loomis, 1995]. Programmers try to develop database applications using object-oriented languages. The object-oriented database management systems (ODBMS) are an integration of object technology within database management systems. ODBMS contains many advantages of an object model, such as flexible type structures, inheritance, reusable codes and data encapsulation, etc. One main advantage of ODBMS is that it can be used to build complex data models without wasting storage space. But ODBMS also has a few small disadvantages. One disadvantage is that it is not convenient or safe to query objects in the databases when using object-oriented languages directly. The Object Database Management Group (ODMG) has defined a query language called Object Query Language (OQL) [Cattell, 1996]. OQL provides declarative access to objects in the object-oriented database. It also supports the same type defined by the application object model. However, the complete implementation of OQL in the small database is difficult and unnecessary.

Summary

This thesis has defined a new object query language which is an extension to C++. The query language is based on collections of objects. It extends the square syntax of collections to take into account the selection predicates. All query items are specified explicitly in a query structure. This kind of query language provides a natural interface to C++ programmers. It has a simple form easy to implement. However, the querying capability is powerful, and it can meet most application needs.

An application of the object-oriented database is implemented completely through the university personnel management database. This database manages the university personnel. It has five data classes: Person, Student, Staff, TA and Professor. Inheritance is the only relationship among these classes. The class Person is a common super type of other classes. The class TA inherits both Student and Staff. The class Professor inherits Staff, but also has its own behavior. The implementation of the personnel database follows the standard of ODMG-93 except the query language. The application program is easy to operate. It has a main menu which provides 9 choices. These choices allow creation and deletion of objects in the database and querying them. The query method is simple. When entering a query statement the program will parse and evaluate the query. If there is no syntax error, the query results will be outputted. If there exists any syntax error the query will be aborted without exiting the program. All these are done at run-time. This system does not use a preprocessor. Querying the objects in the databases can be continued until returning to the main menu. The collections formed by an earlier

query results can be queried again later as long as there has been no exit from the program.

Conclusion

This study provided a new method to query objects in an object-oriented database by defining an object query language extension to C++. The new object query language has a natural interface for C++ programmers. The query structure is based on a bracketed syntax of object collections. The query selection predicates are included within the brackets. This kind of object query language provides the same basic functionality as SQL does. Its querying capability can meet many application's needs.

The query method was implemented successfully through an application of the object-oriented database: "University Personnel Management Database." This application is used in the management of university personnel. The implementation does not use a preprocessor as many methods do. Instead, a query statement is passed to a query execution function as a string parameter at runtime. At that time the query statement is translated and executed. The program provides a simple interface for users, particularly for querying. All queries are done during execution without exiting the program.

Suggested Future Work

There are some restrictions for the new query language. First, it is not complete. Some complicated query functions cannot be achieved by applying the query language. We need to make more extensions. For example, support operation functions of class in the selection predicates, group-by and all operators in the query and get the i^{th} element of

an indexed collection. The extension should be made to support the complicated join query functions. Second, the implementation needs to be improved. An improved implementation would be able to handle several query statements at one time. All these improvements will make the query language more powerful and efficient.

REFERENCES

1. Aho, A., Sethi, R. and Ullman, J. (1986). *Compiler principles, Techniques, and Tools*. Reading, MA: Addison-Wesley, 1986.
2. Cattell, R. G. G., ed. (1996). *The Object Database standard: ODMG-93 (Release 1.2)*. San Francisco, CA: Morgan Kaufmann, 1996.
3. Cattell, R. G. G., ed. (1997). *The Object Database standard: ODMG: 2.0*. San Francisco, CA: Morgan Kaufmann, 1997.
4. Chan, H., Lu, H. and Wei, K. (1993). A Survey of SQL Language. *Journal of Database Management*. 4(4): 4-15. Fall 1993.
5. Fleming, C. and von Halle, B. (1989). *Handbook of Relational Database Design*. Reading, MA: Addison-Wesley, 1989.
6. Jausen, G. and Vossen, G. (1998). *Models and Languages for Object-Oriented Databases*. Reading, MA: Addison-Wesley, 1998.
7. Jordan, D. (1996). ODMG OQL: The Object Query language. *C++ Report*, 8(2). 1996.
8. Khoshafian, S. (1993). *Object-oriented Databases*. NY: John Wiley & Sons, 1993.
9. Kim, W., ed. (1995). *Modern Database Systems*. Reading, MA: Addison-Wesley,

1995.

10. Loomis, M. (1993). Making Objects Persistent. *Journal of Object-Oriented Programming*, 6(6): 25-28. October, 1993.
11. Loomis, M. (1994). Querying Object Database. *Journal of Object-Oriented Programming*, 7(3): 56-78. June, 1994.
12. Loomis, M. (1995). *Object Databases: The Essentials*. Reading, MA: Addison-Wesley, 1995.
13. Lusardi, F. (1988). *The Database Experts' Guide to SQL*. Intertext Publications/Multiscience Press, Inc., 1988.
14. Ullman, J. (1980). *Principles of Database Systems*. Potomac, MD: Computer Science Press, 1980.
15. Ullman, J. (1988). *Principles of Database and Knowledge-base Systems*. Rockville, MD: Computer Science Press, 1988.
16. Vadaparty, K. (1995a). Persistent Pointers: I. *Journal of Object-Oriented Programming*, 8(4): 14-18. July, 1995.
17. Vadaparty, K. (1995b). Revisiting Persistent Pointers. *Journal of Object-Oriented Programming*, 8(5): 62-64. September, 1995.
18. Vadaparty, K. (1995c). Memory-mapped Architectures. *Journal of Object-Oriented Programming*, 8(6): 18-26. October, 1995.

19. Vadaparty, K. (1995d). Pointer Swizzling at Page-fault Time. *Journal of Object-Oriented Programming*, 8(7): 12-20. November - December, 1995.
20. Vadaparty, K. (1996). Developing an ODBMS Application: Basic Steps. *Journal of Object-Oriented Programming*, 8(8): 19-25. January, 1996.
21. Wade, A. E. (1997). Object Query Standards. *Object Magazine Online*. June, 1997.

APPENDIX

APPENDIX A GLOSSARY OF TERMS

attribute A conceptual notion employed to express an identifiable association between the object and some other entity or entities.

behavior The observable effects of performing the requested service.

binding The process of selecting a method to perform a requested service and selecting the data to be accessed by that method.

class Template from which objects can be created. It is used to specify the behavior and attributes common to all objects of the class.

concurrency control The mechanisms are necessary to enforce serializability of transactions. There are four basic modes: pessimistic mode, optimistic mode, mixed mode and semi-optimistic mode.

dynamic binding Binding that is performed at run time, after the request has been issued.

encapsulation The facility by which access to data is restricted to legal access. Illegal access is prohibited in an object by encapsulating the data and providing the member functions as the only means of obtaining access to the stored data.

inheritance The mechanism by which new classes are defined from existing classes. Subclasses inherit operations of their parent class. Inheritance is the mechanism by which reusability is facilitated. It is a mechanism for sharing behavior and attributes between classes. It allows one class to be defined in terms of another class. Objects can

inherit data and methods from other objects. Inheritance helps implement “is-a” or “kind-of” relationships.

integrity A kind of consistency that guaranteed the existence of all objects referenced.

The consistency of the database can be typically expressed through predicates or conditions on the current state of the database.

messaging The process of invoking an operation on an object. In response to a message, the corresponding method is executed in the object. A message to an object specifies what should be done. A message can be sent by clients of the object-application programs, another object, or another method within the same object.

methods Implementations of the operations relevant to a class of objects. The part of an object that performs an operations is termed a method. Methods are invoked in response to messages.

modularization A method to develop and write software in small, understandable modules of data structures and operations allowable on these data.

object A combination of data and the collection of operations that are implemented on the data; also, a collection of operations that shares a state. The representation of a real-world entity. An object is used to model a person, place, thing, or event from the real world. It encapsulates data and operations that can be used to manipulate the data and responds to requests for service.

persistence The ability of data to exist beyond the scope of the program that created it. The phenomenon whereby data outlive the program execution time and exist between executions of a program. All databases support persistence.

persistent object An object whose existence is independent of the lifetime of the creating program.

polymorphism Ability to apply the same operation to different classes of objects. The operation on the object can be invoked without knowing its actual class.

query An activity that involves selecting objects from implicitly or explicitly identified collections based on a specified predicate.

recovery The process of enforcing consistency after a transaction has aborted as a result of the state of certain objects, hardware failures, or communication problems.

request An event consisting of an operation and zero or more actual parameters that causes a service to be performed.

reusability The concept of easily using existing software within new software; the ability to use well-designed software modules that have been tested, in several places, in different applications, so as to minimize development of new code. Object-oriented languages employ inheritance as a mechanism for reusability.

security The mechanisms are necessary to protect operations and objects from illegal or dangerous actions. Access control mechanisms are typically employed to restrict access to objects or operations by other objects, systems, or users. Objects, users, and systems

can be provided with certain capabilities that authorize them to perform certain operations.

signature Definition of the types of the parameters for a given operation.

transaction A sequence of database operations that transforms a consistent state of a database into another consistent state, without necessarily preserving consistency at all intermediate points.

transient object An object whose existence is limited by the lifetime of the process that created it; a temporary object.

type A predicate defined over value that can be used in a signature to restrict a possible parameter or characterize a possible result.

APPENDIX B TABLE OF ACRONYMS

ADTs	Abstract Data Types
ANSI	American National Standards Institute
DBMS	Database Management Systems
E-R	Entity-Relational
ODBMS	Object-oriented Database Management Systems
ODL	Object Definition Language
ODMG	Object Database Management Group
OML	Object Manipulation Language
OQL	Object Query Language
PL	Programming Language
SQL	Structured Query language
TA	Teaching Assistant

VITA

Xijian Tang

Candidate for the Degree of

Master of Science

Thesis: QUERYING OBJECT-ORIENTED DATABASES FROM C++

Major Field: Computer Science

Biographical:

Personal Data: Born in Tongcheng, Anhui, P. R. China, on January 13, 1962, the son of Daihong Tang and Xiuying Feng; married to Weihua Liu in 1986.

Education: Graduated from Kongcheng High School, Tongcheng, Anhui, P. R. China in June, 1979; received Bachelor of Science degree in Petroleum Engineering from Petroleum Institute of Eastern China, Dongying, Shangdong, P. R. China in July 1983; completed the requirements for the Master of Science degree with a major in Computer Science at Oklahoma State University in December, 1998.

Experience: Employed by the Research Institute of Daqing Oil Production Technology in Daqing, Heilongjiang, P. R. China as an Engineer, 1983-1988; Employed by the Petroleum Engineering Department at Texas Tech University in Lubbock, Texas as a Visiting Scholar, 1988-1989; Employed by the Research Institute of Daqing Oil Production Technology in Daqing, Heilongjiang, P. R. China as a Senior Engineer 1990-1996; Employed by the Biosystems and Agricultural Engineering Department at Oklahoma State University in Stillwater, Oklahoma as a research assistant, 1998 to present.