

AN INTERFACE BETWEEN OBJECT-ORIENTED  
DATABASES AND RELATIONAL DATABASES  
USING B-TREE OBJECT-RELATIONAL  
ACCESS INTERFACE

By

QING CHEN

Bachelor of Science

Xian Jiaotong University

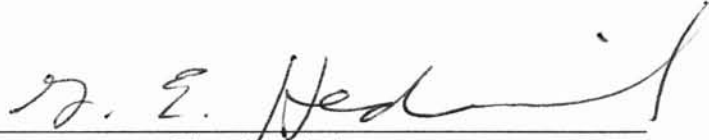
Xian, P. R. China

1994

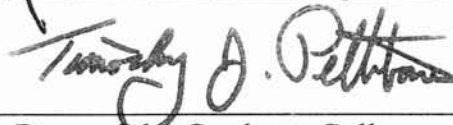
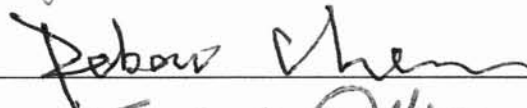
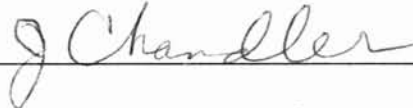
Submitted to the Faculty of the  
Graduate College of the  
Oklahoma State University  
in partial fulfillment of  
the requirements for  
the Degree of  
MASTER OF SCIENCE  
December, 2002

AN INTERFACE BETWEEN OBJECT-ORIENTED  
DATABASES AND RELATIONAL DATABASES  
USING B-TREE OBJECT-RELATIONAL  
ACCESS INTERFACE

Thesis Approved:



Thesis Adviser



Dean of the Graduate College

## ACKNOWLEDGEMENTS

I would like to thank Dr. G. Hedrick for his constant support and advice that made this work possible. I am grateful for his willingness to discuss the work at any time and his thoughtful criticism. I am grateful to other committee members, Dr. John P. Chandler and Dr. Debao Chen for their helpful suggestions, advisement, and readiness to be my co-referee.

I am specially grateful to Dr. Hong, the Vice Present of BarDyne, Inc., both for supporting the design and implementation of the B-Tree database access interface kernel and for providing ideas for future directions of this work. In addition, I would like to give special appreciation to my parents for their strong encouragement at times of difficulty, love, support, and understanding throughout the whole process to implement this thesis.

Finally, I would like to thank the Computer Science Department for supporting me during my studies.

## TABLE OF CONTENTS

Chapter	Page
I. INTRODUCTION .....	1
1.1 Motivation .....	3
1.2 Organization .....	5
II. LITERATURE REVIEW .....	6
2.1 Relational Database Model .....	6
2.2 Relational Database Query Languages .....	12
2.3 Basic Concepts of Object-Oriented Programming .....	14
2.4 Object-Oriented Database .....	17
2.5 Introduction of Object Query Language .....	19
2.6 B-Trees .....	22
III. B-TREE OBJECT-ORIENTED DATABASE ACCESS INTERFACE STRUCTURE AND IMPLEMENTATION .....	32
3.1 B-Tree object-oriented Database Access Interface Structure .....	32
3.2 Implementation of Object-Relational Mapping Layer .....	36
3.3 Implementation of Object B-Tree Layer .....	43
3.4 Implementation of Object-oriented Query Operation Layer .....	51
IV. PERFORMANCE ANALYSIS .....	58
4.1 Platform, database and applications for performance analysis .....	58
4.2 Test Results .....	63
4.3 Performance Test Summary .....	67
V. CONCLUSIONS .....	69
5.1 Advantages vs. Disadvantages .....	69
5.2 Research Contribution .....	71
5.3 Future Work .....	71
BIBLIOGRAPHY .....	73
GLOSSARY .....	76

Chapter	Page
APPENDIX A - Class Schema Definition for B-tree Object-Relational Database	
Access Interface .....	79

## LIST OF TABLES

Table	Page
3.1 Customer Contact Table .....	46
3.2 Products Information Table .....	47
4.1 Definitions of Test Database Table FluidProperty .....	58
4.2 Definitions of Test Database Table FlowData .....	59
4.3 Definitions of Test Database Table PressureData .....	59
4.4 Definitions of Test Database Table TemperatureData .....	59
4.5 Definitions of Test Database Table ViscosityData .....	59
4.6 Definitions of OBT Access Interface of Select Queries .....	62
4.7 Definitions of OBT Access Interface of Join Queries .....	62
4.8 Performances of Select Query (A) .....	63
4.9 Performances of Select Query (B) .....	64
4.10 Performances of Join Query (A) .....	65
4.11 Performances of Join Query (B) .....	66

## LIST OF FIGURES

Figure	Page
1.1 Entity-Relationship Schema .....	2
1.2 The Tabular Representation of E-R Schema .....	3
2.1 The Flow of commands and data between the user and the database .. ....	8
2.2 Union and Difference Relations .....	9
2.3 The Intersection of R and S .....	10
2.4 The Division of R and S .....	10
2.5 The Theta-join of R and S .. .....	11
2.6 The Natural join of R and S .....	12
2.7 Expression of Relational Algebra in SQL .....	14
2.8 Inheritance from super class to sub class .. .....	16
2.9 The structure of a B-tree node .....	23
2.10 The Structure of a B-tree of order 5 .....	24
2.11 B-tree Insertion Step 1 of order 5 .....	25
2.12 B-tree Insertion Step 2 of order 5 .....	25
2.13 B-tree Insertion Step 3 of order 5 .....	25
2.14 B-tree Insertion Step 4 of order 5 .....	26
2.15 B-tree Insertion Step 5 of order 5 .....	26
2.16 B-tree Insertion Step 6 of order 5 .....	27

2.17 B-tree Insertion Step 7 of order 5	27
2.18 B-tree Insertion Step 8 of order 5	28
2.19 B-tree Deletion Step 1 of order 5	28
2.20 B-tree Deletion Step 2 of order 5	29
2.21 B-tree Deletion Step 3 of order 5	30
2.22 B-tree Deletion Step 4 of order 5	30
2.23 B-tree Deletion Step 5 of order 5	31
3.1 B-Tree Object-Oriented Database Access Interface Structure	35
3.2 ODBC components	37
3.3 CMapDB Class Schema Definition	79
3.4 CMapRecordset Class Schema Definition	80
3.5 CMapException Class Schema Definition	81
3.6 Try-Catch for Exception Handling Schema Definition	42
3.7 Try-Catch for Database Mapping Schema Definition	43
3.8 MapBtItemType Class Schema Definition	81
3.9 Schema Definition of Template Class MapDBItem	83
3.10 Schema Definition of Template Class MapBtNodeType	84
3.11 Schema Definition of Template Class MapBtMidNode	85
3.12 Schema Definition of the template class MapBTree	87
3.13 Schema Definition of the class template MapBTreeIterator	88
3.14 Schema Definition of CMapParser	89



3.15	The Pseudo-code for Smart parser .....	54
3.16	The Architecture of classes in B-Tree Object-relational Database Access Interface .....	56
3.17	An Instance of class MapBTree .....	57
4.1	Performance Test Output Screen for table FluidPorperty .....	60
4.2	Performance Test Output Screen for table ViscosityData .....	61
4.3	Performance Curve for Select Query (A) .....	63
4.4	Performance Curve for Select Query (B) .....	64
4.5	Performance Curve for Join Query (A) .....	65
4.6	Performance Curve for Join Query (B) ..	66

## CHAPTER I

### INTRODUCTION

Many information systems use relational database systems for efficient sharing, storage, and retrieval of large quantities of data. The large-scale use of relational database systems can be attributed to the relational model that is easy to understand and has a well understood theoretical foundation. Structured Query Language (SQL) is an integrated database language with strong declarative features that encompass the full range of functionality: data definition, data manipulation, query, data control, transaction management, and database security [Loomis, 1995].

Relational database systems face a growing challenge from object-oriented technology. This technology's attraction results from the semantically rich modeling constructs that give users a natural, flexible, and easily extendable and reusable mechanism to model complex data structures such as those used in multimedia, compound documents, and computer-aided design (CAD) applications. Unlike the relational data model, the object-oriented data model stores all data in the objects which are instances of data classes, Figure 1.1 is an Entity-Relationship (E-R) schema [Ullman, 1988] of an IT company employee management database. Figure 1.2 is the tabular representation of the E-R schema. There is more redundancy shown in Figure 1.2 than in Figure 1.1 because the object model can easily represent this kind of relationship as inheritance. At this point, the relational and object-oriented data models are not, in their current state, compatible. If relational database systems are to become the predominant model for managing not only simple data, but also complex structures, we must develop

special techniques to convert data in a relational database to a format that object-oriented applications can access and manipulate easily.

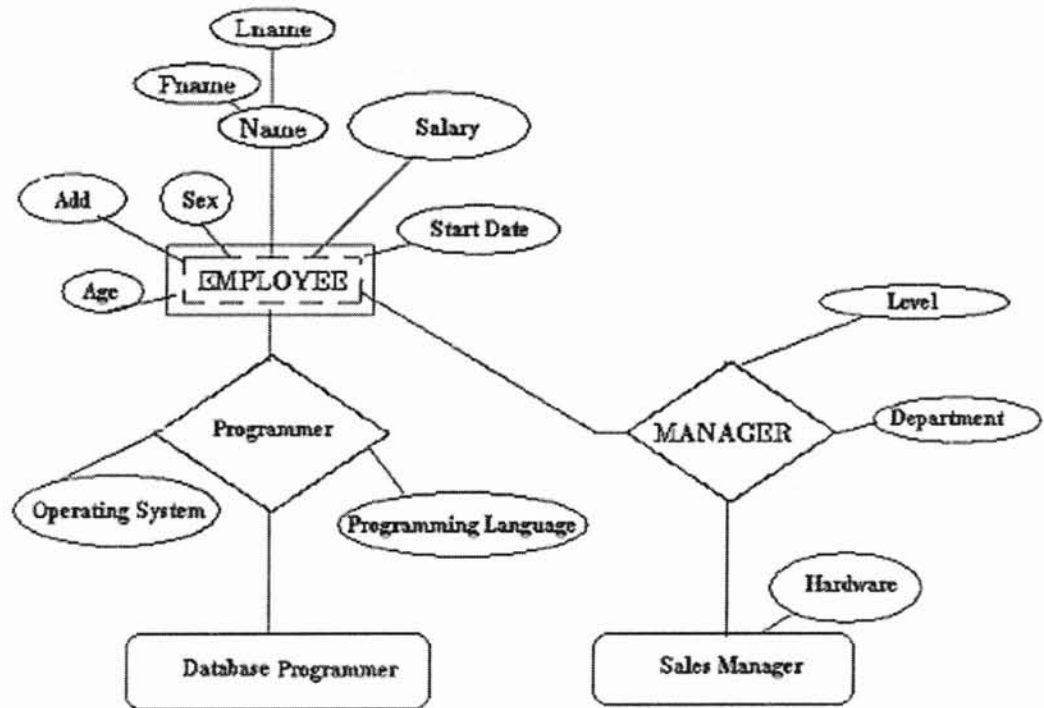


Figure 1.1 Entity-Relationship Schema [Ullman, 1988]

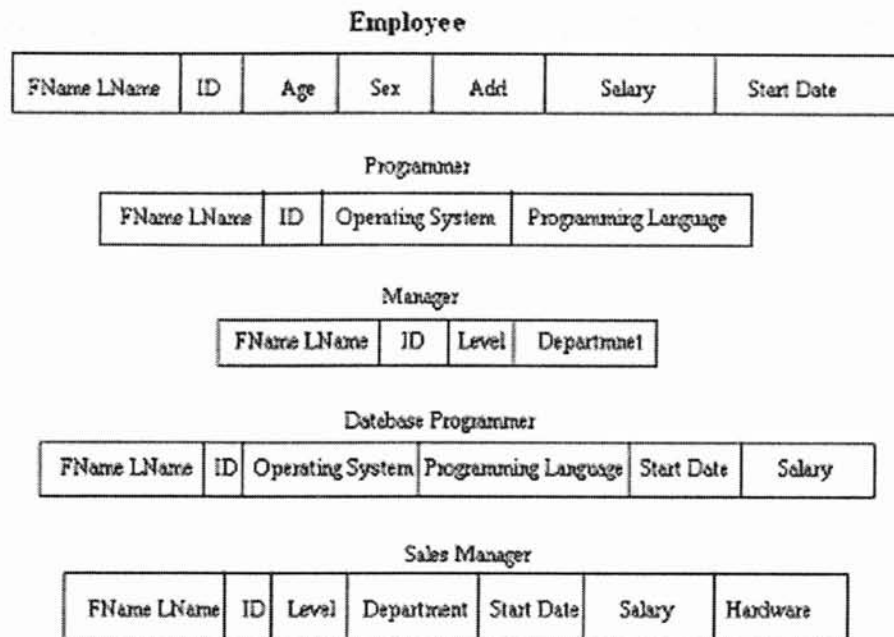


Figure 1.2 The Tabular Representation of E-R Schema

### 1.1 Motivation

In the last a few years it became obvious, to the database industry and its customers that the relational model is neither designed for nor able to cope effectively with the new types of data that it is expected to store. These new types of data can include audio, video and image files as well as user-defined types. Traditional relational databases support data types such as integer, decimal and character plus special data types for date and time, and these relational database systems cannot meet the needs of new data types and new applications developed with object-oriented programming language. On the other hand, object modeling is not standardized, so each programming environment implements its own variation. This motivates one to study how to combine the current relational database system and object-oriented programming to offer benefits of real environment. Fortunately, relational theory is concerned with knowledge and

object techniques are concerned with behavior. Mapping between the two models describes how the two worlds can refer to each other and show how they can be integrated.

This thesis extends traditional B-Tree structures to include a new object-oriented access method --- B-Tree Object-Relational Database Access Interface (Figure 3.1) that extracts object-oriented constructs such as object classes and relationships between object classes (association, aggregation, and other inheritance relationships) from the definition of an existing relational database. The resulting methods provide at least four benefits that can be used in current relational systems:

- The new access algorithm provides for the use of binary data types to store either multi-valued attributes or complex-valued attributes in a relational column, and optimizes forms of the major schemas to reduce cost of data operations that involve multiple tables.
- This approach allows object-oriented access to relational databases with no penalty and minimal overhead for either style of access.
- The underlying database system supports traditional business applications through the standard relational interface, and supports advanced multimedia or compound document applications through the object-oriented interface.
- New advanced object-oriented applications can use existing data in relational databases through tuple-to-object translations.

## 1.2 Organization

The next chapter reviews related work in existing literature. Chapter III introduces the design and implementation of object B-tree Database Access Interface; Chapter IV summarizes the performance of object B-tree Database Access Interface; Finally, Chapter V presents conclusions and suggestions future work to be done in this area.

## CHAPTER II

### LITERATURE REVIEW

#### 2.1 Relational Database Model

The database systems and models can be distinctively categorized under the following three generations of database systems:

1. Hierarchical and Network Databases
2. Relational Databases
3. Object-Oriented, Object-Relational Databases

Except for these three prominent generations of database systems, other types of databases for special purposes have also been designed. Some of the important ones include knowledge-based database systems, temporal database systems, spatial database systems, document databases, and multi-media databases [Ullman, 2001].

The relational model, described in [CODD, 1990], is clearly the model on which most current systems are being designed. In this section, the relational model is presented as background to the future work in the area. No intent is made to provide the same mathematical preciseness as provided in the original presentation, however, examples are given to supplement the discussion.

The relational model is a conceptual data model for representing data objects. The data objects may represent real or abstract objects but have in common the fact that within each relation; each object can be described by a unique set of values taken from a predefined set of domains. The relational model divides naturally into two sections: the data structures, and the rules for data manipulation.

The data structures of the relational model are merely abstract data structures and not necessarily implemented physically [Date, 1995]. That is, the data a user wishes to access or manipulate should appear to respect all of the properties to be described in this section. For example, according to the model, all tuples within the same relation are unique. In distributed systems, however, it may be cost efficient to store several copies of certain records (the physical implementation of a tuple) in various locations permanently. This does not violate any rules of the model as long as the user is not able to detect this no matter what operations he performs on the data. The user should be shielded from all knowledge of how the data actually is stored. For this reason, research in query optimization is extremely important. If a user is supposed to be ignorant of how the data is stored, he cannot be expected to pose his queries in a manner that can be processed efficiently. Thus, it is the job of a query optimizer to ensure that queries to the database are answered in a reasonable amount of time while preserving the intent of the user.

A relational database is a collection of tables that the user can access and manipulate abstractly according to the restrictions set in the relation model. The actual structural implementation may not conform to the relational model [Date, 1995]. A database management system (DBMS) is an application for handling the storage and retrieval of data. Generally, a DBMS provides the following advantages:

- Access control.
- Centralized control.
- Controlled data redundancy.
- Data concurrence and consistency.
- Data independence.



- Data integrity.
- Data maintenance.
- Recovery.

A relational DBMS (RDBMS) is a type of DBMS, which is perceived to hold data in a series of two-dimensional tables. Each table consists of a number of rows (tuples) and columns (attributes). Relationships between rows in the different tables are represented by the storage of foreign keys (A foreign key is a column in a table where that column is a primary key of another table, which means that any data in a foreign key column must have corresponding data in the other table where that column is the primary key) within one table. The method of data access in a RDBMS is through a set of operators in the relational algebra. The relational algebra serves only to manipulate permanently stored relations into temporary relations that may in turn either be reviewed by the user or permanently stored via "update" commands of the data manipulation language (Figure 2.1).

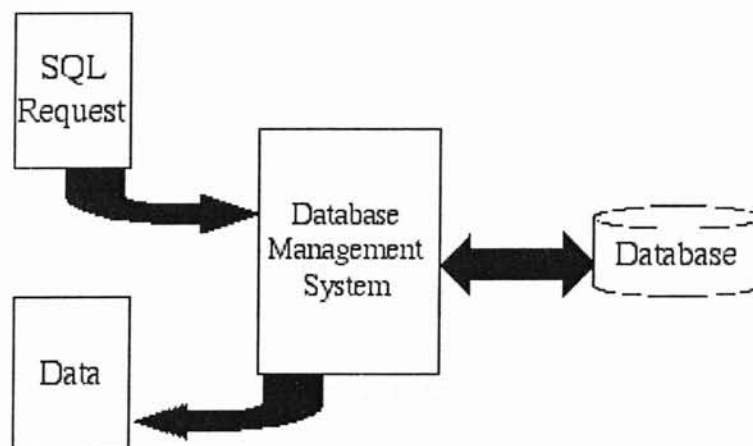


Figure 2.1 The Flow of commands and data between the user and the database

The basis of the relational algebra is the following set of five operators: selection, projection, union, difference, and Cartesian product. Four other commonly used operators that are expressible in terms of the basic operators are: unions, difference, intersection, division, theta-join, and natural-join. To illustrate application of the basic relational operators, Ullman [Ullman, 1988] gives the following relations which result from application of each of the basic operators to the relations R and S from Figure 2.2 to Figure 2.6.

Figure 2.2 describes unions and difference relations. The columns of R and S can have different names as long as the relations have the same number of components. However, the resulting relation has no obvious names for its columns.

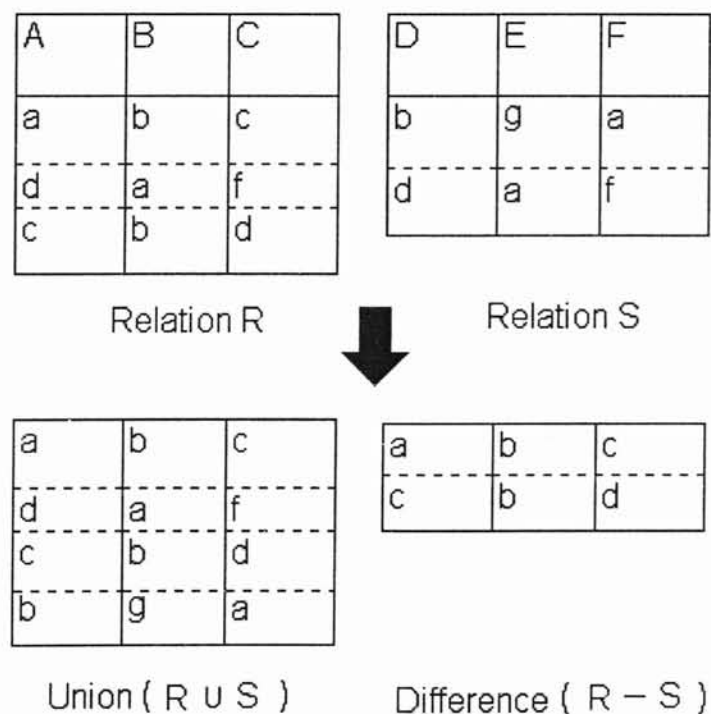


Figure 2.2 Union and Difference Relations

Figure 2.3 describes intersection relation. The intersection of R and S can be expressed as shorthand for  $R - (R - S)$ . A, B, C, D, E and F are the column names of relation R and S.

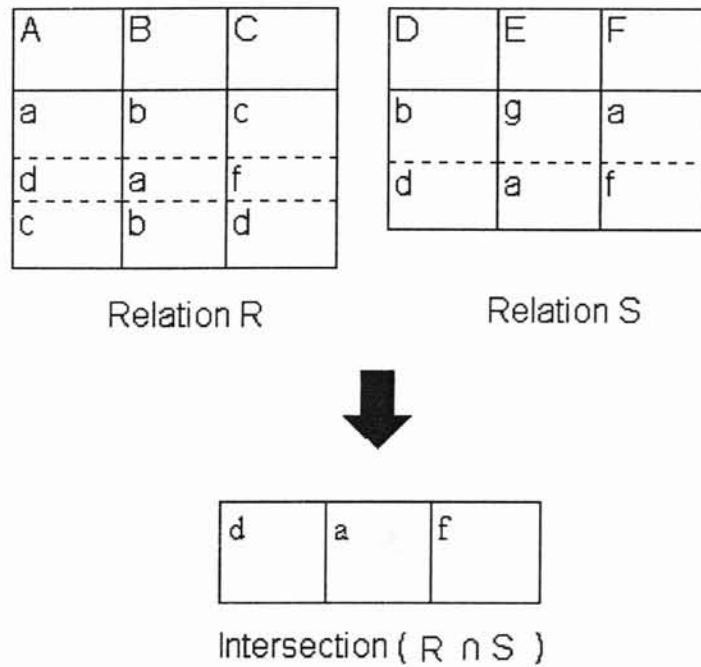


Figure 2.3 The Intersection of R and S

Figure 2.4 describes division relation. Tuple *ab* is in the resulting relation because tuples *abcd* and *abef* are in R, and tuple *ed* is in the resulting relation for a similar reason. Tuple *bc*, which is the only other pair appearing in the first two columns of R, is not in the resulting relation because *bccd* is not in R.

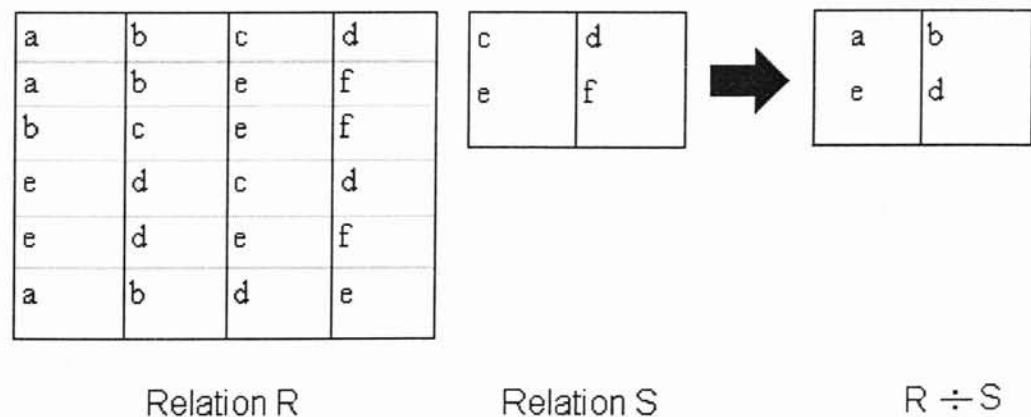


Figure 2.4 The Division of R and S

Figure 2.5 describes theta-join of relation R and S on columns “i” and “j” written  $R \bowtie_{i \theta j} S$ .  $\theta$  (theta) is an arithmetic comparison operator ( $=$ ,  $<$ , and so on). That is, the theta-join of R and S is those tuples in the Cartesian product of R and S such that the  $i^{\text{th}}$  component of R stands in the relation  $\theta$  to the  $j^{\text{th}}$  component of S. In Figure 2.5, “ $\theta$ ” is the arithmetic comparison operator “ $<$ ”, “i” is attribute B and “j” is attribute D.

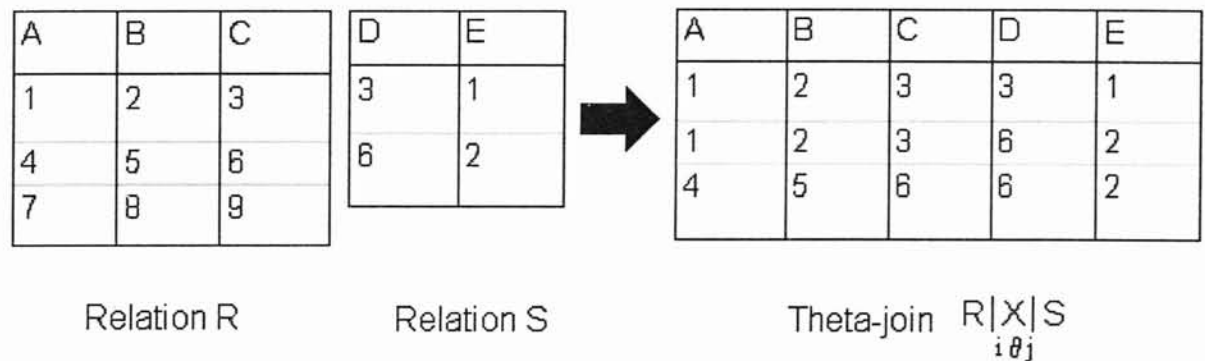


Figure 2.5 The Theta-join of R and S

Figure 2.6 describes natural join relation, written  $R \bowtie S$ . Each tuple in R can be considered to see which tuples of S agree with it in both columns B and C. For example, tuple *abc* in R agrees with *bcd* and *bce* in S, so the resulting relation gets tuples *abcd* and *abce*. Similarly, *dbc* gives *dbcd* and *dbce* for the natural join. Tuple *bbf* agrees with no tuple of S in columns B and C, so natural join obtains no tuple in that begins with *bbf*. Lastly, tuple *cad* matches *adb*, and natural join gets tuple *cadb*.

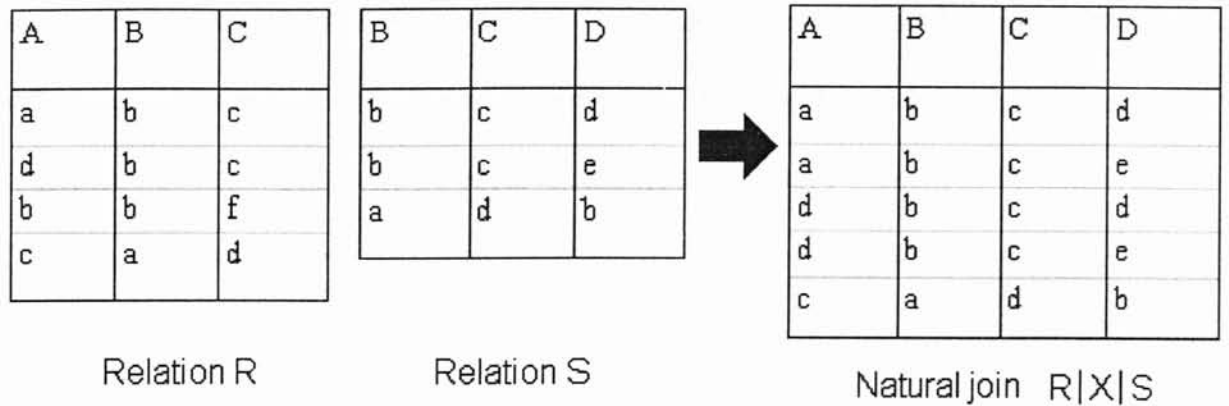


Figure 2.6 The Natural join of R and S

## 2.2 Relational Database Query Languages

Structured Query Language (SQL) is the most popular relational database query language. 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].

In 1974 Donald Chamberlin and others defined the language SEQUEL (Structured English Query Language) at IBM Research [Chamberlin, 1974]. This language was first implemented in an IBM prototype called SEQUEL-XRM in 1974-75. In 1976-77 a revised version of SEQUEL called SEQUEL/2 was defined and the name was changed to SQL subsequently. IBM developed a new prototype called System R in 1977. System R implemented a large subset of SEQUEL/2 (now SQL) and a number of changes were made to SQL during the project. System R was installed in a number of user sites, both internal IBM sites and also some selected customer sites. Thanks to the success and acceptance of System R at those user sites IBM started to develop

commercial products that implemented the SQL language based on the System R technology [Matheus, 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]. The American National Standards Institute (ANSI) adopted it 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 even has prompted the makers of many non-relational database products to provide SQL as a means of access to their product [Halpin, 1995]. 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 mechanisms.

SQL is based on tuple relational algebra and comprised of data definition statements, and data manipulation statements. SQL provides two types of data manipulation statements: database "update" statements and database manipulation statements. UPDATE, DELETE, and INSERT all make changes in permanent storage to the database. A distinctive feature of SQL is that queries can be nested to an arbitrary depth by using SELECT statements in the predicate of the WHERE clause [Date And Darwen, 1993].

SQL can implement all of the basic relational operators. The following table from [Date, 1995] illustrates the implementation of each relational operator in SQL.

$A \cup B$	SELECT * FROM A UNION SELECT * FROM B
$A - B$	SELECT * FROM A WHERE NOT EXISTS (SELECT * FROM B WHERE all-fields-of A = all-fields-of B)
$A \times B$	SELECT * FROM A,B
$\sigma_p(A)$	SELECT * FROM A WHERE p
$\pi_{x,y,\dots,z}(A)$	SELECT DISTINCT x,y,\dots,z FROM A

Figure 2.7 Expression of Relational Algebra in SQL

Thus, techniques that serve to optimize a general SQL expression also optimize a query posed in the relational algebra.

### 2.3 Basic Concepts of Object-Oriented Programming

Object-oriented programming supplies the raw modeling power of objects with the management flexibility of classes and inheritance [Date, 1995]. In the object-oriented paradigm, objects are the atomic units of encapsulation; classes manage the collections of objects; inheritance structures the collection of objects and the inheritance structures of classes.

#### Objects

Complex objects are built from simpler ones by applying constructors to them. The simplest objects are integers, characters, byte strings of any length, boolean values,

and floats (one might add other atomic types). Objects may be attached to, or communicate to another one by way of a well-defined user interface such as class pointer or structure pointer; also, objects may be classified according to common behavior and other characteristics, such as: (1) Abstract data type; (2) Data encapsulation; (3) Inheritance. The object constructors must be orthogonal to the objects; that is, any constructor should apply to any object.

### Class

Class is a prototype that defines the variables and the methods common to all objects of a certain kind. Class includes two parts: the interface and the implementation. Only the interface part is visible to the users of the class; the interface consists of a list of operations together with the data type of their input parameters and the results.

Implementation consists of a data part and an operation part. The data part describes the data type and data structures of class member variables. The operation part consists of procedures that implement the operations of the interface part.

### Inheritance and Class Hierarchies

The concept of inheritance is a second reusability mechanism derived from its parent properties. It lets a class, called a subclass, be defined starting from the definition of other classes and messages. In addition, a subclass can have specific attributes, methods, and messages that are not inherited. Moreover, the subclass can override the definition of the super class's attributes and methods.

A class can have several subclasses. Some systems let a class have several super classes (multiple inheritance), while others impose the restriction of a single super class. Based on inheritance, the set of classes in the schema can be organized in an inheritance



graph (Figure2.8).

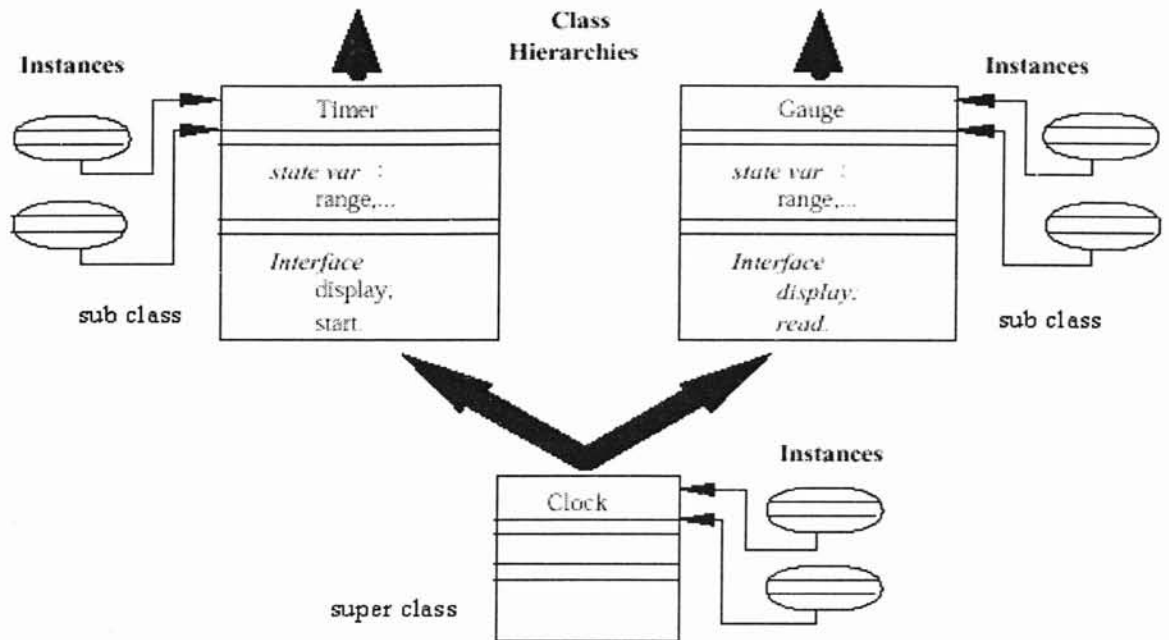


Figure2.8 Inheritance from super class to sub class

### Encapsulation

The idea of encapsulation comes from the need for a clear distinction between the specification and the implementation of an operation, and the need for modularity. Modularity is necessary to structure complex applications designed and implemented by a team of programmers. It is also necessary as a tool for protection and authorization.

The idea of encapsulation in programming language comes from the theory of abstract data types. An object has operations that can be performed on the object. It is the only visible part of the object. The implementation part has a data part and an operation part. The data part is the representation or state of the object, and the operation part describes the implementation of each operation.

## 2.4 Object-Oriented Database

An Object-Oriented Database Management System (OODBMS) stores, shares, and manages objects instead of tables of data as in Relational Database Management Systems (RDBMS). Roughly, “an object oriented database system is one that supports the classical features of database systems”, which were summarized by Ullman as a system that supports “large amounts of persistent, reliable, shared data.” [Ullman, 1988]. Objects in an OODBMS reassemble real-world objects in many ways; they all have states, behaviors, and identities, and the structure and behavior of similar objects can be defined in common classes. An OODBMS maintains the basic functionality of a traditional database management system, such as persistency, concurrency, and recoverability, by transparently integrating database capabilities with an object-oriented programming language such as C++ or SmallTalk. Unlike a RDBMS that stores simple and often fixed length data in tables and thus has difficulties in representing complex relations, an OODBMS can easily support complex structures naturally by using objects. Therefore an OODBMS is the “combination of object-oriented analysis (OOA) and design (OOD), object-oriented programming language (OOP), and object-oriented database (OODB) offers the benefits of a synergistic development environment” [Loomis, 1995]. OODBMS features include:

### Persistence

The ability of the programmer to have the data survives the execution of a process. In OODB, each object is allowed to become persistent independent of its type without explicit casting.

### Secondary Storage Management

The performance features (such as index management, data clustering, data buffering, access-path selection, and query optimization) hidden to the user.

### Concurrency

The ability of the DBMS to offer all users working simultaneously the same level of service

### Recovery

The ability enables software to bring itself back to some coherent state of the data in case of hardware or software failure.

Object-oriented (OO) features include:

### Complex Objects

Complex objects are built from simpler ones by applying constructors to them. Constructors must be orthogonal to the objects (i.e., they can be applied to every object). The use of complex objects improves the capability of representing of the real world.

### Object Identity

OO systems are identity-based, meaning that each representation of information has its own identifiers. Identity-based model are common in OO programming language but rather new in DB technology, since n most relational DB, relations are valued based.

### Encapsulation

An object contains both programs and data and has both an interface and an implementation part. The interface part is the specification of the set of operations that

can be performed on the object; the implementation part describes the implementation of each operation. In most OODB, even data specification is part of the interface.

### Types and Classes

A type summarizes the common features of a set of objects; a class has an extension which contains a set of operations and class instances using which the user can manipulate the objects.

### Class or Type Hierarchies

Inheritance is the ability of a subclass to receive all data and operations coming from its super-classes. It helps to build reusable and better-structured code, presents more concise description of the real world and the shared specifications of applications.

### Overriding

Overloading and Late Binding: when a single identifier is bounded to different operation codes in different types, then the code is overridden and the operation is side to be overloaded. To provide this functionality, code is not bound to an operation identifier at compile time but at run time, performing late binding.

## 2.5 Introduction of Object Query Language

Object Query Language (OQL) is an object database query language for querying and manipulating object databases that conform to the Object Data Model, and is specified as a part of the ODMG (Object Data Management Group) standards-93 [Cattell, 1997]. OQL is an object database sub-language in that the sort of queries and

manipulations are expressed in the object algebra. OQL is a declarative (as opposed to procedural) language that allows ODBMS seamless operations and algebraic transformation: expressions factorization, priority to selections and constant expressions.

The features of OQL are listed as follows:

- 1) OQL is used to describe the retrieval and manipulation of objects.
- 2) The Object Definition Language (ODL) defines interfaces to object types used by the OQL code;
- 3) OQL does not provide a full programming environment, but it can be imbedded in a programming language and invoke programming language functions
- 4) The object-oriented programming languages (C++, Smalltalk, and Java) are used to implement applications programs, with embedded OQL statements to manipulate the object database.

"The designers of OQL have adopted an evolutionary approach. Rather than designing a completely new language, they have based OQL on SQL, the standard language for the relational databases." [Cattell, 1997]. Many queries in SQL are also valid in OQL, though sometimes with minor changes, and have a similar meaning. However, OQL also extends SQL to deal with object-oriented notions, such as complex structured objects, object identity, and relational data model. The design of OQL is functional and the results of queries have types, which allow them to be queried again. The syntax of OQL based on SQL, the queries can return:

- An object
- A collection of objects
- A literal
- A collection of literals

For example:

```
select distinct struct(t: title, d: department) from people x where  
x.title="programmer"
```

This builds a structure containing title and department and returns a literal of type `set<struct>`. The OQL expressions can take one of the two forms: implicit or explicit. The explicit form uses the select clause, while the implicit one does not use it. For instance:

```
Person.name = "fonda";
```

This is an implicit expression that returns all Person instances whose name is "fonda".

```
select x.name from x in Person where x.name = "fonda";
```

This is an explicit expression that returns all Person instances whose name is "fonda". A particular limitation of OQL is that "it cannot express arbitrarily complex computations.

It is therefore necessary to use OQL in conjunction with other computer languages".

There are other limitations when the OQL is used to make complex queries, such as joins [Jordan, 1998].

## 2.6 B-Trees

A B-tree is a particular type of tree-structured index. Since its introduction by Bayer and McCreight (1972), there have been many variations on the original theme and it is the index structure that has become the most popular for structuring the physical database. Most objected-relational databases offer some variation of the B-tree while some provide no other option [Date, 1994]. The popularity of the B-tree stems from the fact that its associated algorithms for inserting and deleting a key value ensure that the tree remains balanced. Balanced trees are those where the leaf nodes are the same distance (i.e. the same number of levels) away from the root node. Other tree structures become unbalanced over a period of time and as a consequence have unpredictable search path lengths since accessing a multi-level index requires accessing one node on each level.

Each node in a B-tree contains pointers that act as delimiters of key values. The structure of a node is shown in Figure 2.9 where  $P_0$  to  $P_n$  are pointers to sub-trees and  $K_0$  to  $K_n$  are key values that appear in ascending order. Nodes of the original B-tree also contain the addresses of the records of the corresponding key values [Wirth, 1986]. A B-tree of order  $m$  has no more than  $m$  pointers in each node (i.e.  $n \leq m$ ). Following a middle pointer will lead to values between the value on its left and the value on its right. Following the leftmost pointer  $P_0$  will lead to values less than  $K_0$  and following the rightmost pointer will lead to values greater than  $K_{n-1}$ .

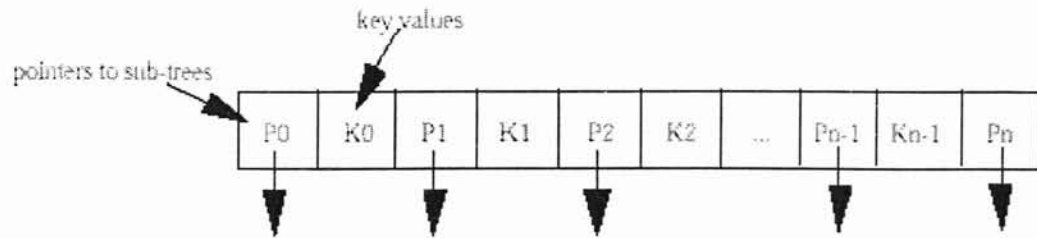


Figure 2.9 The Structure of a B-tree node

The definition of a B-tree of order  $m$  is:

- All leaves are on the bottom level.
- All internal nodes (except the root node) have at least  $\lceil m / 2 \rceil$  children.
- The root node can have as few as 2 children if it is an internal node, and can obviously have no children if the root node is a leaf (that is, the whole tree consists only of the root node).
- Each leaf node must contain at least  $\lceil m / 2 \rceil - 1$  keys.

$\lceil x \rceil$  is the ceiling function. Its value is the smallest integer that is greater than or equal to  $x$ . Thus  $\lceil 3 \rceil = 3$ ,  $\lceil 3.35 \rceil = 4$ ,  $\lceil 1.98 \rceil = 2$ , etc [Crotzer, 1975].

The following is an example of a B-tree of order 5 (Figure 2.10). This means that (other than the root node) all internal nodes have at least  $\text{ceil}(5 / 2) = \lceil 2.5 \rceil = 3$  children (and hence at least 2 keys). The maximum number of children in a node is 5 (so that 4 is the maximum number of keys).



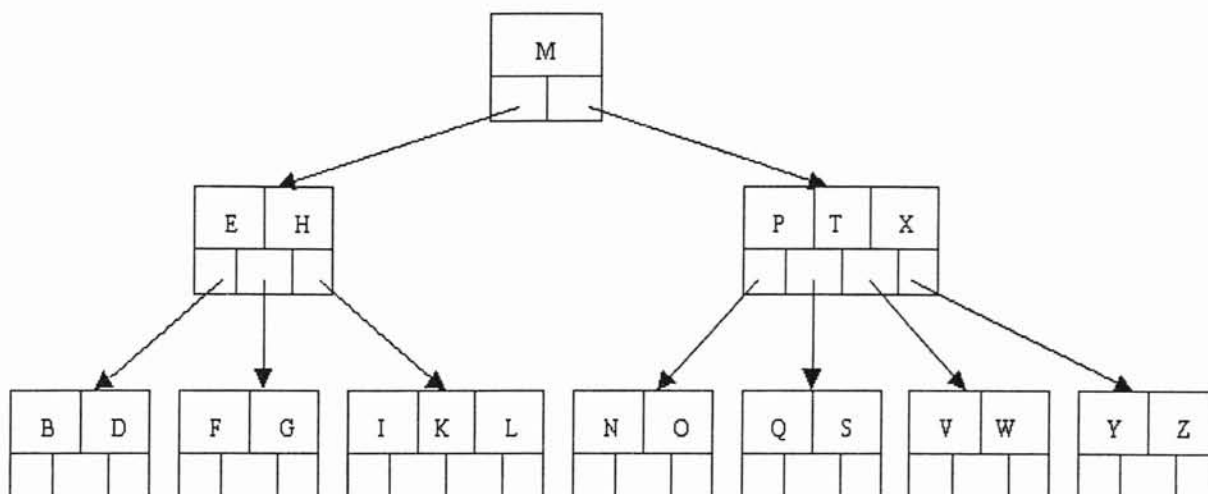


Figure 2.10 The Structure of a B-tree of order 5

The typical operations on a B-Tree include insertion, deletion and split. The insertion algorithm proceeds as follows: When inserting an item, first make a search for it in the B-tree. If the item is not already in the B-tree, this unsuccessful search will end at a leaf. If there is room in this leaf, just insert the new item here. This may requires some existing keys be moved one to the right to make room for the new item. If instead this leaf node is full so that there is no room to add the new item, then the node must be split with about half of the keys going into a new node to the right of this one. The median (middle) key is moved up into the parent node. (If that node has no room, then it may have to be split as well.) When adding to an internal node, not only might we must move some keys one position to the right, but the associated pointers have to be moved right as well. If the root node is ever split, the median key moves up into a new root node, thus causing the tree to increase in height by one.

An example of Insertion: Insert the following letters into what is originally an empty B-tree of order 5: A G F B K D H M J E S I R X C L N T U P. Order 5 means that a node can have a maximum of 5 children and 4 keys. All nodes other than the root must

have a minimum of 2 keys. The first 4 letters get inserted into the same node (Figure 2.11).

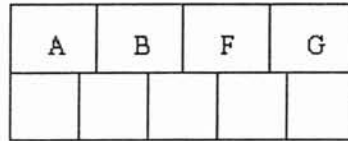


Figure 2.11 B-tree Insertion Step 1 of order 5

When try to insert the K, we find no room in this node, so we split it into 2 nodes, moving the median item F up into a new root node (Figure 2.12).

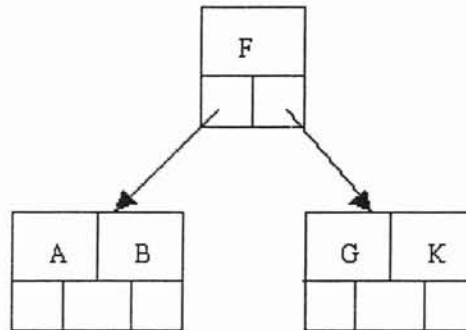


Figure 2.12 B-tree Insertion Step 2 of order 5

Inserting D, H, and M proceeds without requiring any splits (Figure 2.13).

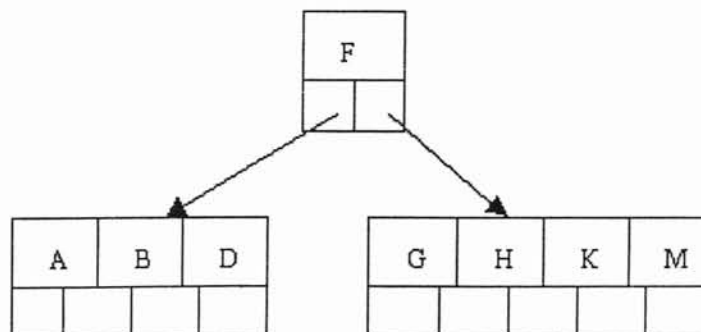


Figure 2.13 B-tree Insertion Step 3 of order 5

Inserting J requires a split. J is the median key, and so it is moved up into the parent node (Figure 2.14).

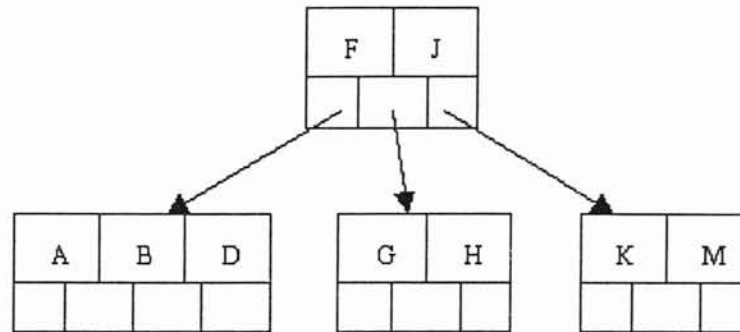


Figure 2.14 B-tree Insertion Step 4 of order 5

The letters E, S, R and I are then added without needing any split (Figure 2.15).

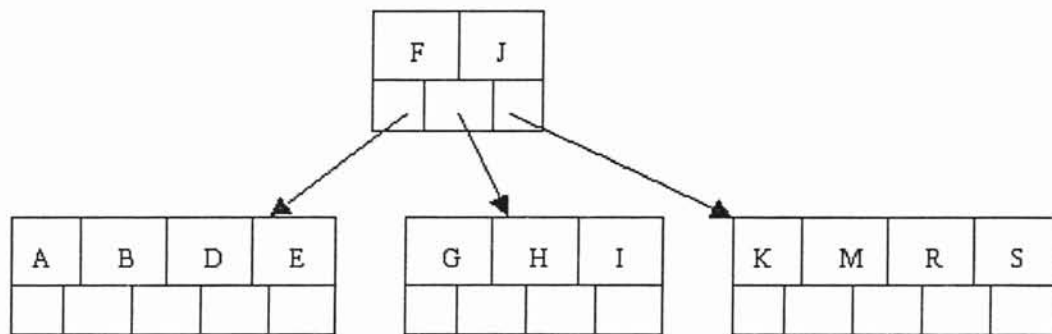


Figure 2.15 B-tree Insertion Step 5 of order 5

When X is added, the rightmost leaf must be split. The median item R is moved up into the parent node. By moving up the median key, the tree is kept balanced, with 2 keys in each of the resulting nodes (Figure 2.16).

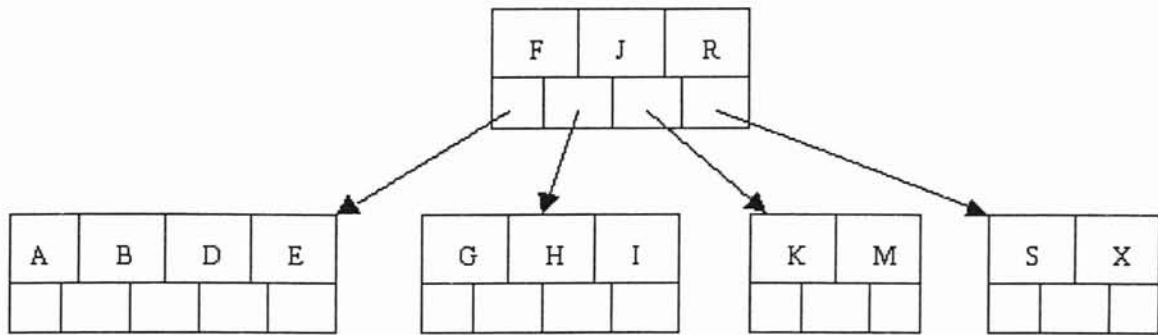


Figure 2.16 B-tree Insertion Step 6 of order 5

The insertion of C causes the leftmost leaf to be split. C happens to be the median key and so is the one moved up into the parent node. The letters L, N, T, and U are then added without any need of splitting (Figure 2.17).

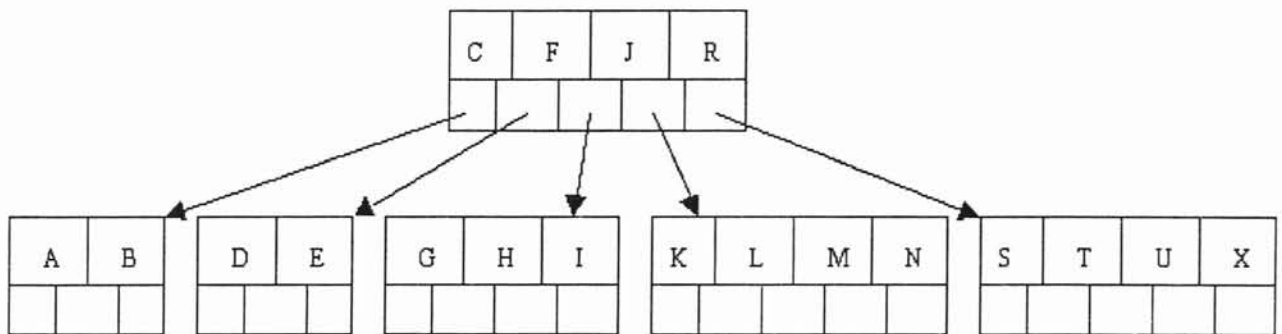


Figure 2.17 B-tree Insertion Step 7 of order 5

Finally, when P is added, the node with K, L, M, and N splits sending the median M up to the parents. However, the parent node is full, so it splits, sending the median J up to form a new root node. Note how the 3 pointers from the old parent node stay in the revised node that contains C and F (Figure 2.18).

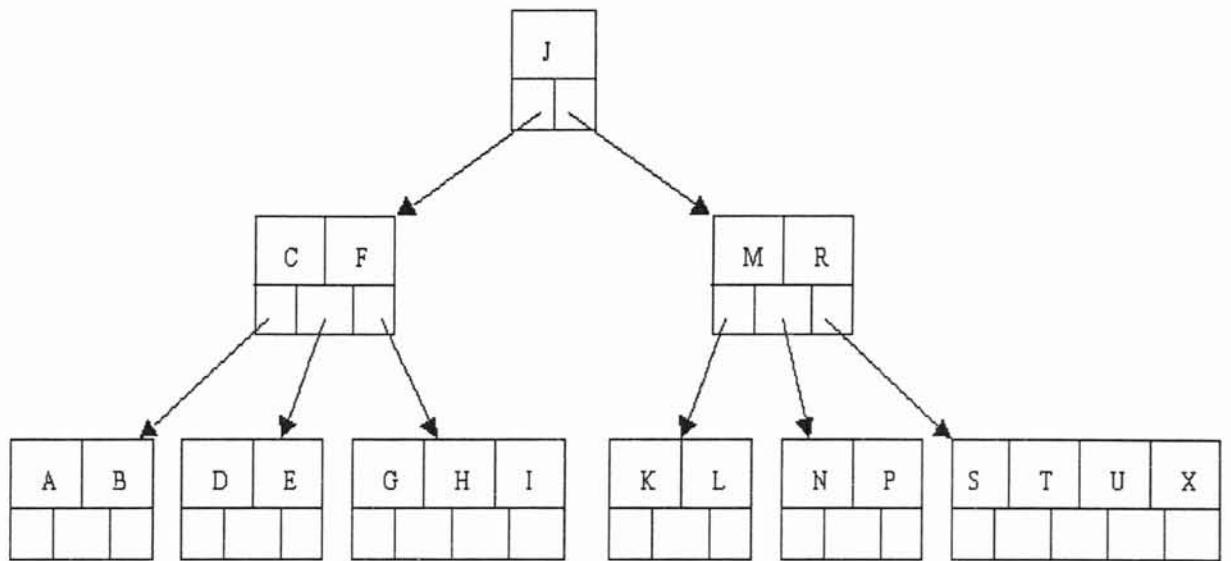


Figure 2.18 B-tree Insertion Step 8 of order 5

An example of Deletion: delete H from the B-tree we built above. At first, we do a lookup to find H. Since H is in a leaf and the leaf has more than the minimum number of keys, this causes to move I over where the H had been (Figure 2.19).

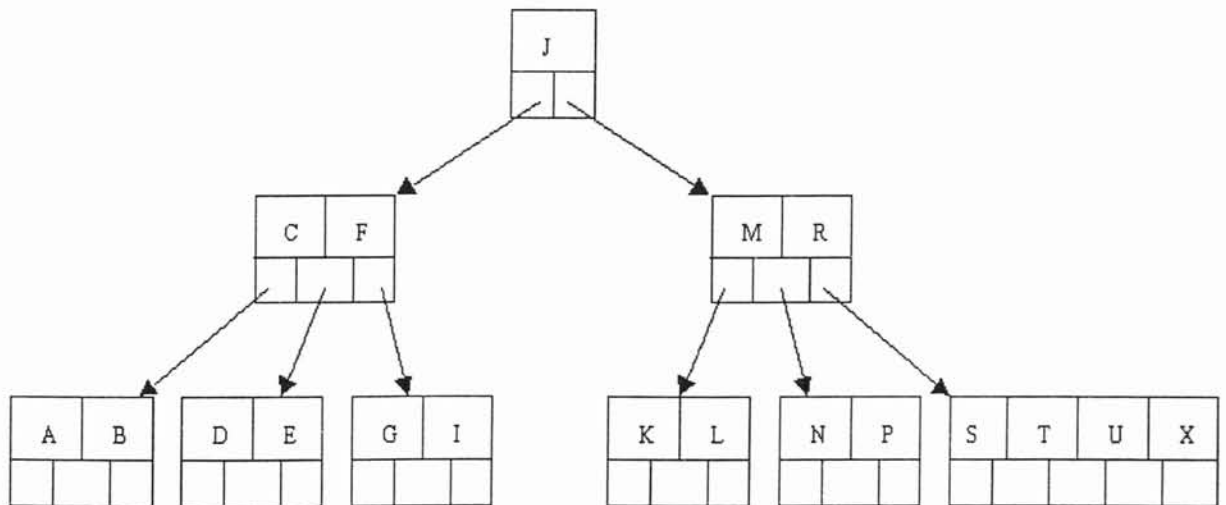


Figure 2.19 B-tree Deletion Step 1 of order 5

Next, delete the R. Since R is not in a leaf, its successor (the next item in ascending order), which happens to be S, and move S up to replace the R (delete S from the leaf) (Figure 2.20).

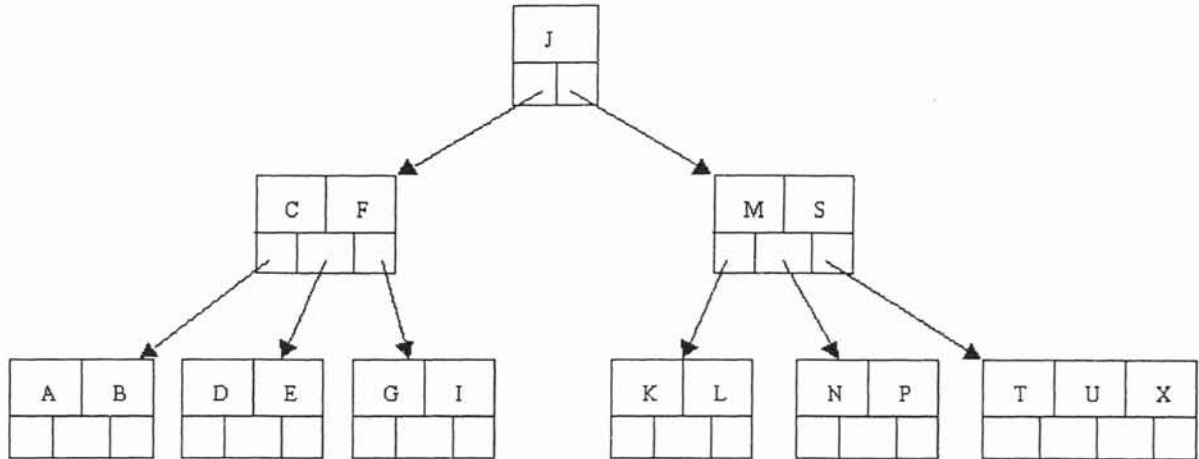


Figure 2.20 B-tree Deletion Step 2 of order 5

Next, delete P. Although P is in a leaf, this leaf does not have an extra key; the deletion results in a node with only one key, which is not acceptable for a B-tree of order 5. If the sibling node to the immediate left or right has an extra key, we can then borrow a key from the parent and move a key up from this sibling. In our specific case, the sibling to the right has an extra key. So, the successor of P, which is S, is moved down from the parent, and the T is moved up (Figure 2.21).

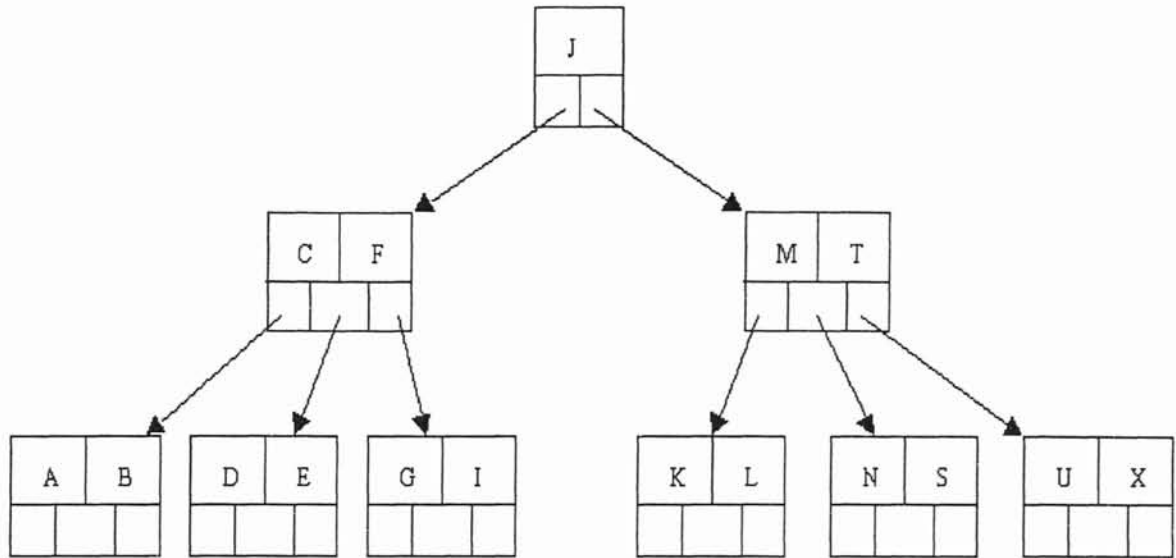


Figure 2.21 B-tree Deletion Step 3 of order 5

Finally, delete D. This causes several problems. Although D is in a leaf, the leaf has no extra keys, nor do the siblings to the immediate right or left. In such a case the leaf must be combined with one of these two siblings. This includes moving down the parent's key that was between those of these two leaves. In this example, it is to combine the leaf containing E with the leaf containing A and B and move down the C (Figure 2.22).

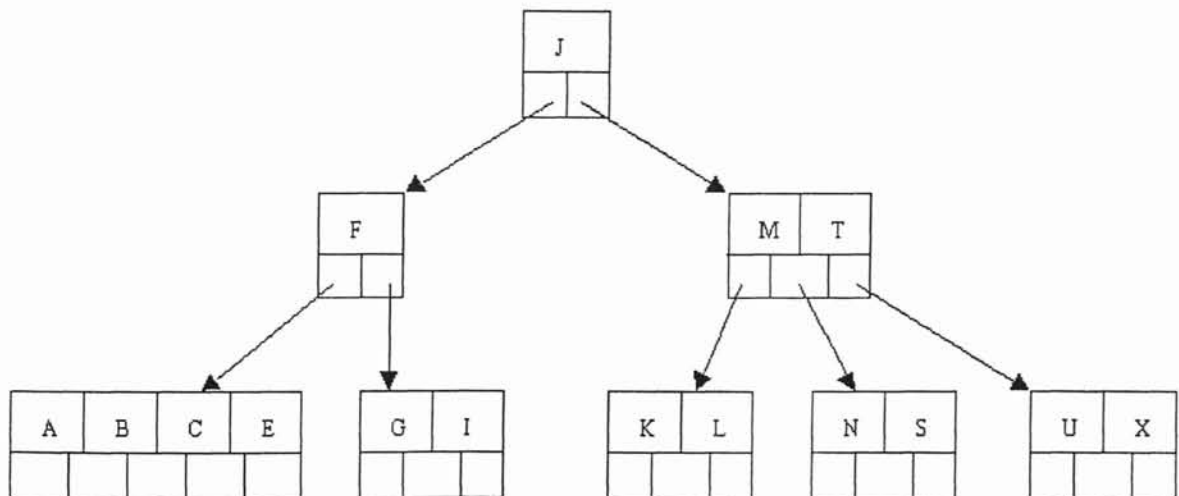


Figure 2.22 B-tree Deletion Step 4 of order 5

Then the parent node contains only one key, F. This is not acceptable. If this node had a sibling to its immediate left or right that had a spare key, then we would again "borrow" a key. However, in this example, there is no way to borrow a key from a sibling, we must again combine with the sibling, and move down the J from the parent. In this case, the tree shrinks in height by one (Figure 2.23).

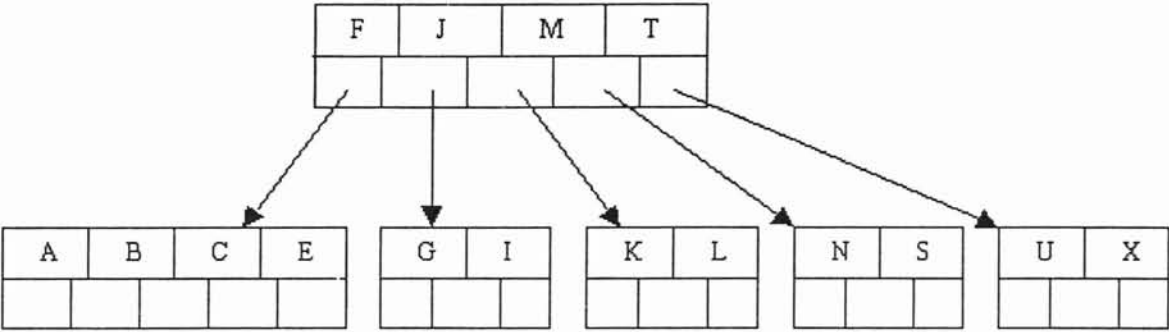


Figure 2.23 B-tree Deletion Step 5 of order 5



## CHAPTER III

### B-TREE OBJECT-RELATIONAL DATABASE ACCESS INTERFACE STRUCTURE AND IMPLEMENTATION

Chapter III describes work that relates to the design and implementation of a B-tree Object-Relational Database Access Interface (BORDBI). First, section 3.1 describes the structure of a BOODAI; Section 3.2 introduces the implementation of database-object mapping because its construction is the first step of building a BORDBI. Then it describes how objects are mapped into a persistent address space since the mapping methods determine features of an object store. Section 3.3 describes how objects can become B-tree nodes in main memory because memory-residency is the key feature of a BORDBI. At the end of this chapter, section 3.4 will introduce how to parse an SQL query and retrieve records from relational database.

#### 3.1 B-tree Object-relational Database Access Interface Structure

The B-tree object-relational database access interface (Figure 3.1) is defined in object-oriented class to achieve a mapping interface to the relational tables. This interface includes three layers:

- Object-oriented Query Operation Layer
- Object B-Tree Implementation Layer
- Object-relational Mapping Layer

The top layer is the Object-oriented Query Operation Layer. It uses an Object Organizer that acts as a project manager. First, the Object Organizer in the database access interface checks with the Object B-Tree Implementation Layer to determine whether the object (B-tree node) already exists in memory. If it does, then Object Organizer returns this object; Otherwise, it passes SQL query to back-end relational database. Within the database access interface the Object Organizer registers all persistent objects existing in memory using a B-tree pointer array, which hold all B-tree's memory address. Registration is done during the execution of object 's constructor. Database access is performed only when necessary.

The middle layer is the Object B-Tree Implementation layer. All objects are organized into the B-Tree structure allowing insertion, update, selection and deletion of objects. The object's properties and behaviors include:

Object Status: Every persistent object has a unique status that is managed by Persistent Object in the B-Tree structure. The object's status determines whether and how the object is written to the database at the end of a database transaction. Each object has one of the following states:

- Active: The object has been retrieved from the database but has not been changed since then. At the end of the database transaction the object is deleted from memory without writing on the database;
- New: The object has been constructed since the last commit statement so that it must be inserted into the database;
- Modified: The object has been modified. At the end of the database transaction the corresponding data are to be updated on the database;

- Deleted: The object has been marked as deleted so that the corresponding data must be deleted on the database;
- New Deleted: The object has been constructed and then been marked as deleted. At the end of the database transaction no database action has to be performed;

Object Delete: Ensures the object's deletion from the database.

Object Read (input parameter): Returns the persistent object to which the table refers.

Object Write (including insert, update, delete): The object is written on the database according to the object's status. This method uses the virtual methods such as insert, update and delete that are implemented for every different subclass of the class persistent object.

Object Mark Modified: Marks the object as modified. This guarantees that the object will be written on the database if it has been modified before committing the database transaction.

Object B-Tree: Converts memory-resident objects into B-Tree nodes; Uses the relational primary key as a B-tree search index to Search, Delete, Insert and Update B-Tree nodes (objects).

The bottom layer is the Object-relational mapping layer. It supports the transition from object references in the application to foreign keys in the database and vice versa. As tuples are retrieved from the back-end relational database, object-relational mapping class functions will convert them into memory-resident objects and load these objects into an object array, and then pass these objects to Object B-Tree Implementation layer to form B-tree nodes. The object that is responsible for writing the relationship to the database is defined as either the relationship source object or the target object. Usually it

is the source object that manages the relationship between the database and the application. On the database, foreign keys at the target object represent relationships. In this case, the source object requests the target object to write the relationship on the database.

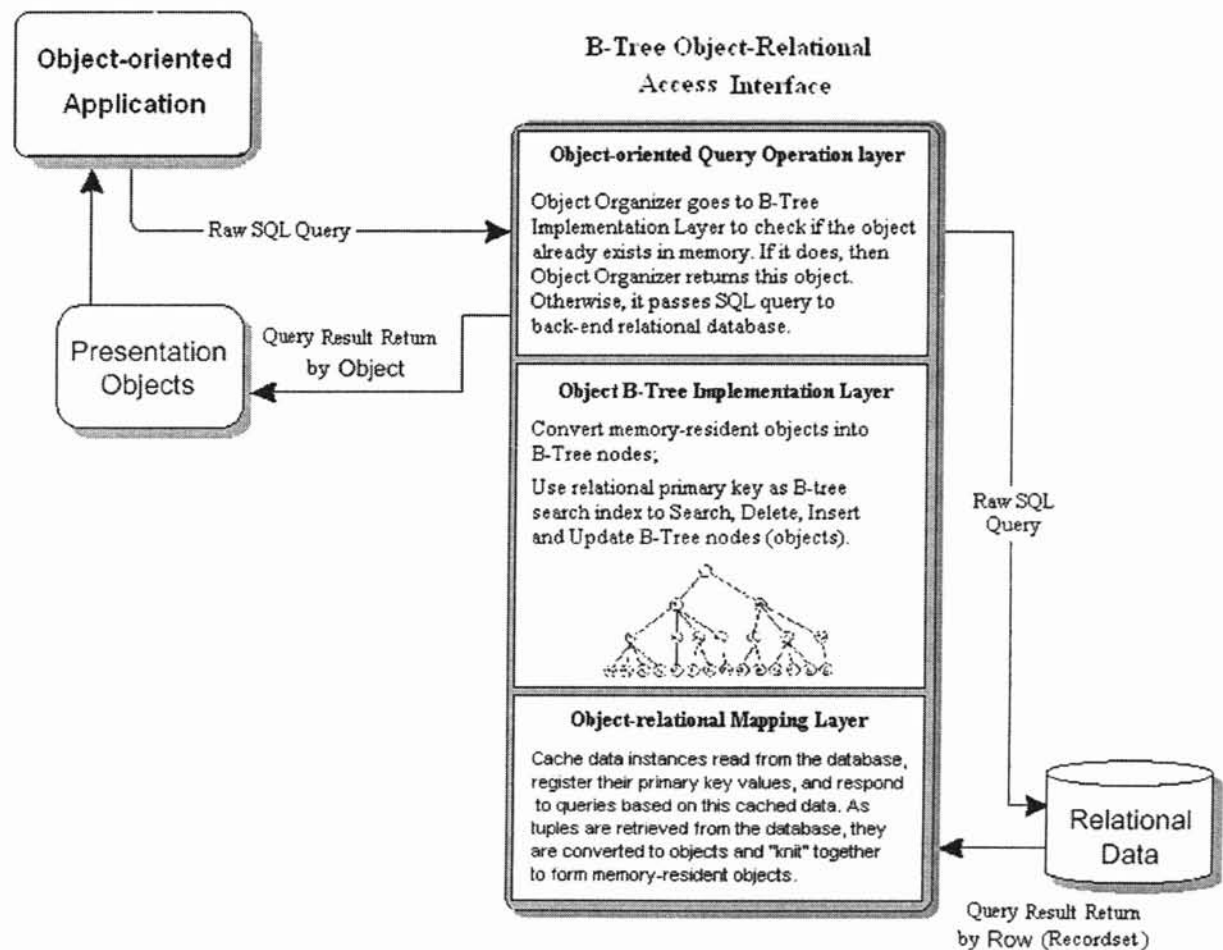


Figure 3.1 B-Tree Object-Relational Database Access Interface Structure

### 3.2 Implementation of Object-relational Mapping Layer

The Database Object-Relational Mapping Layer is composite of a set of classes to provide a framework for using ODBC API to manipulate database.

The Microsoft® Open Database Connectivity (ODBC) is a C programming language interface that makes it possible to access relational databases with SQL statements. ODBC allows Windows-based desktop applications need to be independent of the source database without rewriting the application for each database. The following are components of ODBC (Figure 3.2):

#### ODBC API (Application Programming Interface)

A library of function calls, a set of error codes, and a standard Structured Query Language (SQL) syntax for accessing database.

#### **ODBC Driver Manager**

A Dynamic-Link Library (ODBC32.DLL) that loads ODBC database drivers on behalf of an application. This DLL is transparent to the application.

#### **ODBC database drivers**

One or more DLLs that process ODBC function calls for specific database.

#### **ODBC Administrator**

A tool used for configuring a database to make it available as a data source for an application.

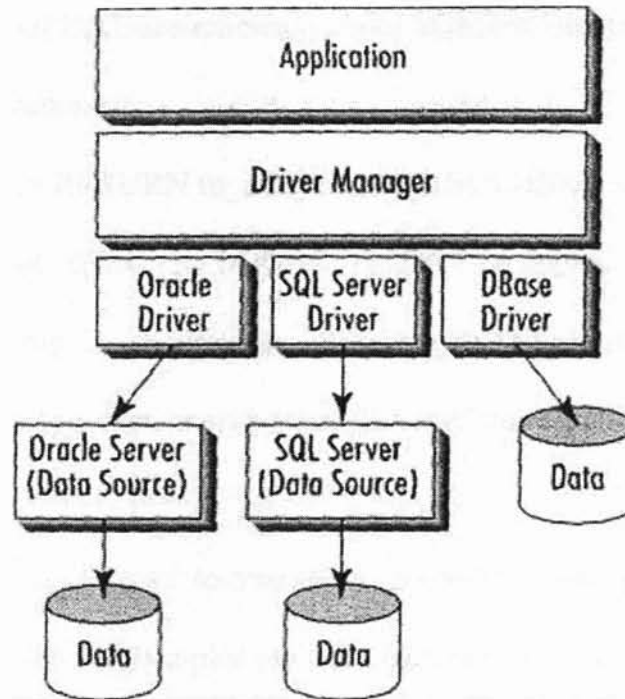


Figure 3.2 ODBC components [Simon, 1996]

The object-relational Mapping Layer consists of three classes: **CMapDB**, **CMapRecordset** and **CMapException** (Figure 3.3, 3.4 and 3.5, Appendix A), which are designed for database open, SQL query, database information retrieve and tuple-to-object conversion respectively. The function members of each class include constructor, destructor and member variables and functions associated with each class. The detailed class information are listed as following:

Name: **CMapDB**

Member Functions: CMapDB(); virtual ~CMapDB();

bool DBOpen(); bool DBClose(); bool SQLDBQuery();

boolIsDBOpen();

void ThrowDBError(); SQLRETURN GetDBReturnCode();

```
bool DBTransactions(); bool SQLDBCommit(); bool  
SQLDBRollback();
```

Member Variables: SQLRETURN m\_hReturnCode; SQLHENV m\_hEnv;

```
SQLHDBC m_hODBC; HSTMT m_hDesc; string m_strDSN;  
string m_strUserName; string m_strPWD; string m_strQueryReturn;  
bool m_bIsOpen; bool m_BAutoCommit;
```

Class Description:

Class CMapDB is designed to represent an object-relational mapping to the back-end relational database. Examples include Microsoft SQL Server, Microsoft Access, Sybase, and Oracle. Class CMapDB calls following ODBC API functions in its member functions:

SQLConnect, SQLDisconnect, SQLAllocHandle, SQLFreeHandle, SQLExecDirect, SQLTransact, SQLEndTran, SQLGetInfo [Simon, 1996]. Before an application can connect to a database, the ODBC environment must be initialized.

CMapDB (SQLAllocHandle) allocates the memory environment and returns a connection handler to the application;

~ CMapDB (SQLFreeHandle) releases memory environment handler when Class

CMapDB is unloaded from memory;

DBOpen (SQLConnect) builds database connection by a data source name (DSN), a user ID (UID) and a password (PWD).

SQLDisconnect terminates a data source connection.

SQLGetInfo retrieves information about the ODBC driver or data source associated with an ODBC connection.

SQLDBQuery (SQLExecDirect) compiles and executes an SQL statement.

DBTransactions (SQLTransact), SQLDBCommit and SQLDBRollback make a commit or rollback operation for all statements associated with a connection. SQLEndTran stops a commit or rollback operation.

By using above member functions, CMapDB can operate on the back-end database with SQL. CMapDB also supports multi-thread so that front-end application can get a connection to one or more database objects active at a time.

Name: **CMapRecordset**

Member Functions: CMapRecordset(); CMapRecordset( CMapDB& db );

virtual ~CMapRecordset(); void ResetContent();

void operator <<( const char\* statement );bool

ExecMapInsertSQL();

bool ExecMapUpdateSQL(); bool ExecMapDeleteSQL();

bool ExecMapSelectSQL(); bool ExecMapFetch();

void SetMapDBOrderBy( ); void SetMapDBGROUPBy();

void SetMapDBWhere();void ThrowError();

const char\* SQLGetValue( ); int GetColumnDataType();

int GetColumnNumber(); void SetHDBC(); bool

MapSQLAllocStmt();

void MapSQLFreeStmt( );

Member Variables: SQLRETURN m\_hReturnCode; SQLHDBC m\_hODBC;

SQLHSTMT m\_hStatement;



### Class Description:

Class CMapRecordset represents a set of records selected from a database. This class can process raw SQL statements, so it can make a SQL query to any kind of relational database table without creating a derived application-specific class from it. CMapRecordset uses a pointer of CMapDB object to access database.

~ CMapRecordset releases all memory resources for database tables and SQL query.

ExecMapSelectSQL selects data from the database table.

ExecMapFetch dynamically fetches table fields at run time rather than statically binding them at design time.

SQLGetValue retrieves field data in the current record (tuple), it supports following data type: string, date, double, integer and long. Using this function, we can build B-tree nodes, which will be described later.

ExecMapInsertSQL adds the new data on the database;

ExecMapUpdateSQL saves the edited data;

ExecMapDeleteSQL deletes the current record from the database.

SetMapDBOrderBy and SetMapDBGROUPBy control how the records are sorted according to a

SQL ORDER BY or GROUP BY clause.

SetMapDBWhere filter those records that meet certain criteria according to a SQL WHERE clause.

GetColumnNumber returns the number of fields in a CMapRecordset object.

m\_hStatement contains the ODBC statement of a CMapRecordset object.

m\_hODBC is an ODBC handler for the database.

Name: **CMapException**

Member Functions: CMapException()

```
CMapException( SQLRETURN return ) { m_nRetCode = return; }
```

```
~CMapException(); Delete();
```

Member Variables: SQLRETURN m\_nRetCode; string m\_strError; string

m\_strStatement;

```
bool m_bExceptionAgain;
```

Class Description:

Class CMapException is designed for CMapDB and CMapRecordset error handling. The C++ language has built-in exception (error) handling abilities. With C++ exception handling, applications can recover from abnormal events. These exceptions are handled by code that is outside the normal flow of control. Exception handlers must delete the exception objects that they handle, because any kind of failure to delete the exception causes a memory leak, in another words, the system resources used for class CMapDB and CMapRecordset are not released from main memory. The structure for C++ exception handling is represented by the following compound-statement syntax:

[Simon, 1996]

**try-block :**

```
try compound-statement handler-list
```

handler-list :

```
handler handler-listopt
```

handler :

**catch-block:** ( exception-declaration ) compound-statement

```

exception-declaration :
    type-specifier-list declarator
    type-specifier-list abstract-declarator
    type-specifier-list
    ...
throw-expression :
    throw assignment-expressionopt

```

Figure 3.6 Try-Catch for Exception Handling Schema Definition

The Figure 3.7 shows the use of CMapException to catch and delete CMapDB and CMapRecordset exceptions. Statement "**try**" builds a try block in which we can open a database connection or execute SQL statements might throw an exception within a try block. Statement "**catch**" builds a catch block and places CMapException code into a catch block. The code in the catch block is executed only if the "**try**" block throws an exception.

```

...
try
{
    ...
    CMapDB.open();
    ...
    CMapRecordset.ExecMapSelectSQL(SQL statement);
    ....
}
catch( CMapException* pMapException )
{
    if (m_bExceptionAgain)
        throw;
    else

    {
        cout<< pMapException->m_strStatement<<endl;
        pMapException->Delete();
    }
}

```

Figure 3.7 Try-Catch for Database Mapping Schema Definition

### 3.3 Implementation of Object B-Tree Layer

The Object B-Tree (OBT) Layer implements an interface that organizes memory-resident objects (database tuples) into the nodes of a B-tree hierarchical structure. An OBT is a database table; the tuples belonging to a table are composed of the attribute values, became the items in its corresponding node. Each node in an OBT is considered to be a collection of several items (keys) and sub-nodes (children). This introduces the problem of 'varying columns' in a table that means different tables in a database may contain different information from each other; each tuple may have a different set of attribute types. Therefore, different classes of objects will be created for their own OBT. Each OBT holds the data and attribute information of a database table. An OBT Layer is a B-tree forest including all kinds of OBT with different structures.

The definitions of the OBT concepts are based on the multi-way search tree algorithms. A multi-way search tree of order  $m$  has to fulfill the following conditions related to the ordering of the keys:

- **The keys in each node are in ascending order.**
- **At every given node the following is true:**

- **The subtree starting at record Node.Branch[0] has only keys that are less than Node.Key[0].**
- **The subtree starting at record Node.Branch[n] has only keys that are greater than Node.Key[n-1] and at the same time less than Node.Key[n]. (0<n<m)**
- **The subtree starting at record Node.Branch[m] has only keys that are greater than Node.Key[m-1].**

If less than the full number of keys is in the node, these conditions are truncated so that they speak of the appropriate number of keys and branches. [Shaffer, 1997]

A B-tree is declared to be of a certain **order** (In the testing program of this thesis, **m=3** is default). This number determines the number of keys contained in any interior node of the tree. Each interior node will contain **order** keys, and therefore **order+1** pointers to sub-trees. A B-tree of order **m** is a multi-way search tree of order **m** such that:

- **All leaves are on the bottom level.**
- **All internal nodes (except the root node) have at least  $\text{ceil}(m / 2)$  (nonempty) children.**
- **The root node can have as few as 2 children if it is an internal node, and can obviously have no children if the root node is a leaf (that is, the whole tree consists only of the root node).**
- **Each leaf node must contain at least  $\text{ceil}(m / 2) - 1$  keys.**

Ceil (x) is the ceiling function. Its value is the smallest integer that is greater than or equal to x. [Weiss, 1997]

The Object B-Tree Layer is consisted of six classes (templates), they describe the implementation of an OBT: how to build an item (key) in the OBT node, how to build an OBT node and how to build an OBT. The detail class (templates) information is listed as follows:

Name: **MapBtItemType**

Member Functions: Figure 3.8 (Appendix A)

Member Variables: Figure 3.8 (Appendix A)

Class Description:

MapBtItemType defines the abstract class structure of an item (key) stored in the OBT node; it is a super-class of the sub-class MapDBItem. MapDBItem can use all virtual functions inherited from class MapBtItemType to reload some specific functions for itself.

Name: **MapDBItem**

Member Functions: Figure 3.9 (Appendix A)

Member Variables: Figure 3.9 (Appendix A)

Class Description:

MapDBItem is a class template derived from class MapBtItemType, defines the real item (key) structure stored in the OBT node according to the attributes of relational tables. An instance of class MapDBItem is an item (key) in the OBT node that contains tuple information. MapDBItem can express any kind of relational table structure. For example, Table 3.1 is suppliers' contact information of a company:

Supplier ID	Contact Name	Contact Title	Address	City	Region	Country	Phone	Fax
1	Charlotte Cooper	Purchasing Manager	49 Gilbert St.	London		UK	011-44-(171) 555-2222	
2	Shelley Burke	Order Administrator	P.O. Box 78934	New Orleans	LA	USA	(100) 555-4822	
3	Regina Murphy	Sales Representative	707 Oxford Rd.	Ann Arbor	MI	USA	(313) 555-5735	(313) 555-3349

Table 3.1 Customer Contact Table

The data structure for Table 3.1 is:

```
typedef struct
{
    int ID;
    string Name;
    string Title;
    string Address;
    string City;
    string Region;
    string PostCode;
    string Country;
    string Phone;
    string Fax;
} SupplierType;
```

The instantiation of Class MapDBItem for Table 3.1 is:

```
SupplierType SupplierTable;
```

```
MapDBItem < int, SupplierType > Supplier(SupplierTable.ID,  
SupplierTable);
```

Table 3.2 is a products information table of the above company:

Product ID	Product Name	Supplier	Category	Unit Price	Units In Stock	Units On Order	Discontinued
1	Chai	Exotic Liquids	Beverages	\$18.00	39	0	No
2	Chang	Exotic Liquids	Beverages	\$19.00	17	40	Yes
3	Aniseed Syrup	Exotic Liquids	Condiments	\$10.00	13	70	No
4	Chef Anton's Cajun Seasoning	New Orleans Cajun Delights	Condiments	\$22.00	53	0	No

Table 3.2 Products Information Table

The data structure for Table 3.2 is:

```
typedef struct  
{  
    int PID;  
    string PName;  
    string Supplier;  
    string Category;  
    float UnitPrice;  
    int UnitsInStock;  
    int UnitsOnOrder;  
    bool Discontinued;  
} ProductType;
```



The instantiation of Class MapDBItem for Table 3.2 is:

```
ProductType ProductTable;
```

```
MapDBItem < string, ProductType > Product(ProductTable.PName,  
ProductTable);
```

Name: **MapBtNodeType**

Member Functions: Figure 3.10 (Appendix A)

Member Variables: Figure 3.10 (Appendix A)

Class Description:

MapBtNodeType is a class template defines abstract class structure for an OBT node according to the relational tables; it is a super-class of the sub-class MapBtMidNode. MapBtMidNode can use all virtual functions inherited from class MapBtNodeType to reload some specific functions for itself.

Name: **MapBtMidNode**

Member Functions: Figure 3.11 (Appendix A)

Member Variables: Figure 3.11 (Appendix A)

Class Description:

MapBtMidNode is a class template derived from class template MapBtNodeType, defines actual interior node class for an OBT according to the relational table structure;

an instance of class template MapBtMidNode is an OBT node in which it contains items (MapDBItem) and pointers of sub-tree (MapBtMidNode).

Name: **MapBTree**

Member Functions: Figure 3.12 (Appendix A)

Member Variables: Figure 3.12 (Appendix A)

Class Description:

MapBTree encapsulates object B-tree algorithms; the algorithms implemented in this thesis assume that items are stored in the internal nodes as well as in the leaves. MapBTree includes member functions to manage nodes (append, delete, insert and update) and B-tree (split, rotate, merge and balance).

Name: **MapBTreeIterator**

Member Functions: Figure 3.13 (Appendix A)

Member Variables: Figure 3.13 (Appendix A)

Class Description:

MapBTreeIterator allows inspection of the items in an OBT node in ascending order according to item index (primary key).

Based on above class templates, the OBT layer provides an object-oriented interface to the relational schema in the back-end database. The main purposes of OBT layer includes:

1. Minimize back-end relational database queries and requests. In order to minimize the total access number to the relational database, an OBT layer should be able to send all the record changes at once instead of updating database as soon as each attribute is changed.
2. Maximize the back-end relational database response abilities and performance. Because the OBT layer is located in the main memory, the relational database stays in the secondary storage devices (disk access). Hence, queries and updates can be processed much faster and more efficiently in the OBT layer than in the database.

Efficient object management in memory and navigation is the key to realize those two purposes. In the OBT layer, as soon as basic tuple information (a record set) is retrieved and passed by the Object-relational Mapping Layer using the primary key to make an SQL query, it is converted to objects. Next their primary key is registered as the index of a B-tree node key (item), and stored into the node as an item according to the B-tree insertion algorithm. Each item in the OBT has one of three status: **Active**, **Update** and **Delete**. These status indicate whether the item is experiencing “**reading**”, “**updating**” or “**deleting**” operations within an OBT without accessing the database each time. OBT will just read the item of “**Active**”; If an item’s status is set as “**Delete**”, then the OBT still holds this item in main memory, treats it as garbage and doesn’t delete it until database transaction is committed, this feature decreases B-tree balance, merge and split operations in the memory and simplifies the object in-memory management. “**Update**” status makes OBT notice database to update records. This approach provides a way to optimize database query performance.

Using the primary key as an OBT item index will make a faster information query and search than the corresponding operation done in the database. Once the data has been retrieved from the database, in-memory navigational queries (B-tree search) allow efficient use of cached object (item/node) information. In order to take advantage of this query support, appropriate indexes should be built to columns that are to be used for class and relationship queries. In general, indexes to be built include all primary and foreign keys in the database schema.

OBT layer also has multi-threaded capabilities to perform background non-blocking query processing. This approach optimizes the database system response time. However, the disadvantages of the above OBT features are the system hardware requirements. In order to keep lots of many database table tuples (sometimes the number is very huge, 100,000 ~ 200,000 tuples) in the memory, the larger main memory, the better the performance is obtained. A multi-threaded B-tree query needs high-speed processors in the system. About ten years ago, the memory chip and CPU were very expensive, the size and speed were very small and slow, it's very hard to implement an OBT layer, but today, hardware components are good enough to develop large scale object systems efficiently.

### 3.4 Implementation of Object-oriented Query Operation Layer

The Object-oriented Query Operation (OQO) Layer provides an interface between Object-oriented application and relational database. The key objective of this layer is

how to query and update database efficiently. In order to achieve this objective, OQO layer employs the following phases:

1. It builds a smart SQL parser (Figure 3.14, Appendix A). This parser recursively recognizes standard SQL key words: **Select, From, Where, Update, Set, Delete, Insert, Order by, Group by**, etc. Using these key words, the parser can decide from a SQL statement the table names, the field names, and how the query is. Next, it contacts with the OBT layer to make an SQL-like B-tree query; following pseudo-code (Figure 3.15) shows how the smart parser recursively recognizes “**Select-From-Where**” clause.

```
#define MAXNUM      50
#define RAWSQL      0x0A00
#define GETTABLE    0x0A20
#define GETFIELD    0x0A22
#define GETWHERE    0x0A24
.
.
.

int nTCounter=0,nFCounter=0;
string m_strSQLStatement, strTableName[MAXNUM], strFieldName[MAXNUM];
string m_strQuery, m_strWhereField, m_strWhereValue;

void OnParserSQL(UINT nFlag) {

    int nL, nP, nP1, nP2;
    nL = m_strSQLStatement.GetLength();

    switch(nFlag) {
    case RAWSQL:
        break;
        .
        .
        .

    case GETFIELD:
        nP1=m_strSQLStatement.Find ("[");
        nP2=m_strSQLStatement.Find ("]");
```

```

        if (nP2-nP1>0) {
            strFieldName[nFCounter++] = m_strSQLStatement.Left (nP2-nP1);
            m_strSQLStatement= m_strSQLStatement.Right (nL-nP2);
            OnParserSQL(GETFIELD);
        }
        else if (nFCounter>0) OnParserSQL(RAWSQL);
        else Throw Error Message("No Field Found!");
        return;

case GETTABLE:
    nP1 = m_strSQLStatement.Find (",");
    nL = m_strSQLStatement.GetLength();

    if (nP1>0) {
        strTableName[nTCounter++] = m_strSQLStatement.Left (nP1);
        m_strSQLStatement= m_strSQLStatement.Right (nL-nP1);
        OnParserSQL(GETTABLE);
    }
    else if (nTCounter>0) OnParserSQL(RAWSQL);
    else Throw Error Message("No Table Found!");

    return;

case GETWHERE:
    nP=m_strSQLStatement.Find ("And");
    if (nP!=-1) OnParserSQL(GETWHERE);

    nP=m_strSQLStatement.Find ("<>" or "<=" or ">=" or "<" or ">" or "LIKE");

    if (nP!=-1) {
        m_strWhereField = m_strSQLStatement.Left (nP);
        m_strWhereValue = m_strSQLStatement.Left (nL-nP);
    }
    else Throw Error Message("Incorrect 'Where' statement!");
    .
    .
    .

default:
    return;

}

if (m_strSQLStatement=="") return;
else if (nP=m_strSQLStatement.Find ("Select")==0) {

    if (m_strQuery == "") m_strQuery = "Select";
    else {
        Throw Error Message("Incorrect 'Select' statement!");
        Return;
    }
}

```

```

        m_strSQLStatement= m_strSQLStatement.Right (nL-nP);

        OnParserSQL(GETFIELD);
    }
else if(nP=m_strSQLStatement.Find ("From")==0)    {

        m_strSQLStatement= m_strSQLStatement.Right (nL-nP);
        OnParserSQL(GETTABLE);
    }
else if(nP=m_strSQLStatement.Find ("Where")==0)    {
        m_strSQLStatement= m_strSQLStatement.Right (nL-nP);
        OnParserSQL(GETWHERE);
    }
    .
    .
    .

else Throw Error Message("Unrecognized SQL statement!");
}

```

Figure 3.15 The Pseudo-code for Smart parser

2. Process OBT layer query. A two-step algorithm processes the OBT query. The first step finds the OBT layer to determine which B-tree is holding the table. If the B-tree is found, then it performs a post-order, depth-first search of the tree using the node index (primary key), to perform necessary accesses and to expose each operation node to the formats of the tuples upon which it is to operate. If none of the trees is found, the smart parser passes that original SQL statement directly to the back-end database to do a query. The second step of the algorithm involves a series of tree traversals that passes the results of the searches up the tree. When a B-tree search is executed to retrieve the tuple information from the item (object) stored in the node (Figure 3.17), a filter is constructed to restrict the results from select and project nodes immediately precede the node, if they exist.

3. The result of a search is an array of items (tuples). The array is converted into a collection of tuples: attribute values are placed in the tuple structures, missing attributes in the items are converted to the relational value NULL. The result is stored in main memory as a list that is accessed in a manner similar to accessing a database table. Performance analysis of the select and join operations are described in Chapter 4. Usually, a **From** clause specifies a single table; however the smart parser must determine all table names, fields name and corresponding database operations for the join queries. This can deteriorate query efficiency.
4. If the smart parser cannot find the correct tree in the OBT layer, as mentioned in the step 2, then the original SQL statement is passed to database directly to make a query. As soon as the query finished, the bottom layer, **Object-relational Mapping Layer**, transforms all tuples into the objects and notice middle layer, **Object B-Tree Layer**, to build an OBT in the memory to store those objects, then send a message to the smart parser to make another OBT layer query again.



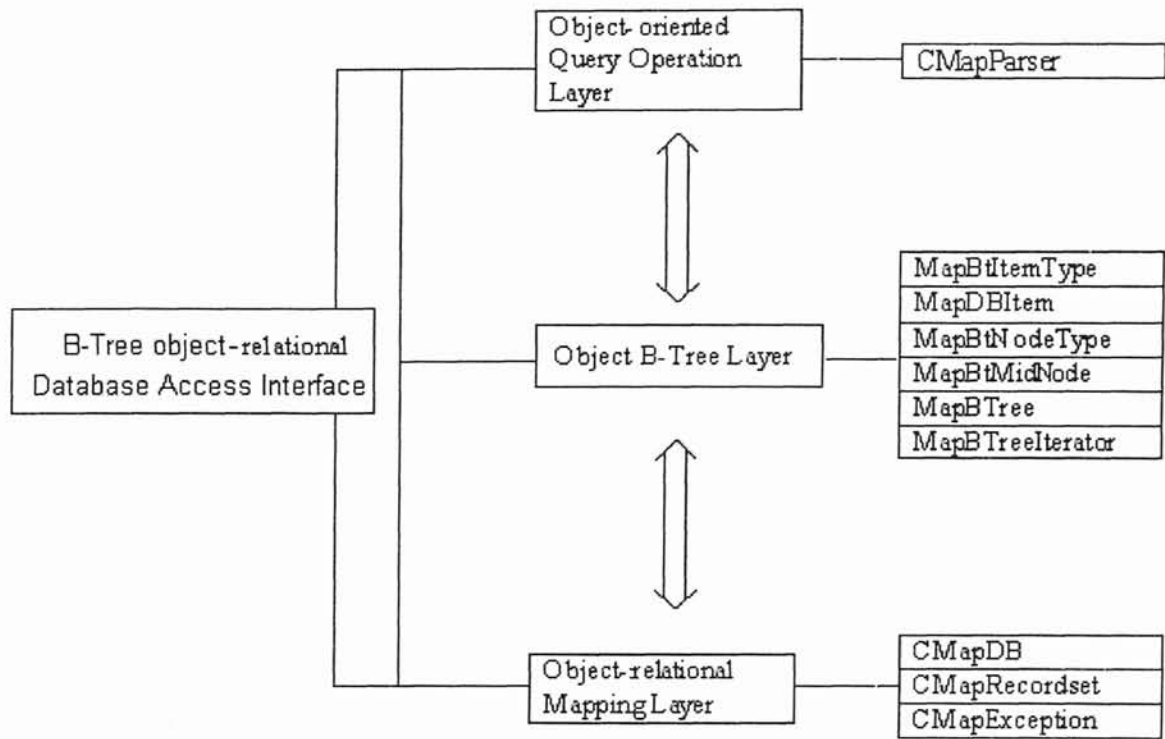


Figure 3.16 The Architecture of classes in B-Tree Object-relational Database Access Interface

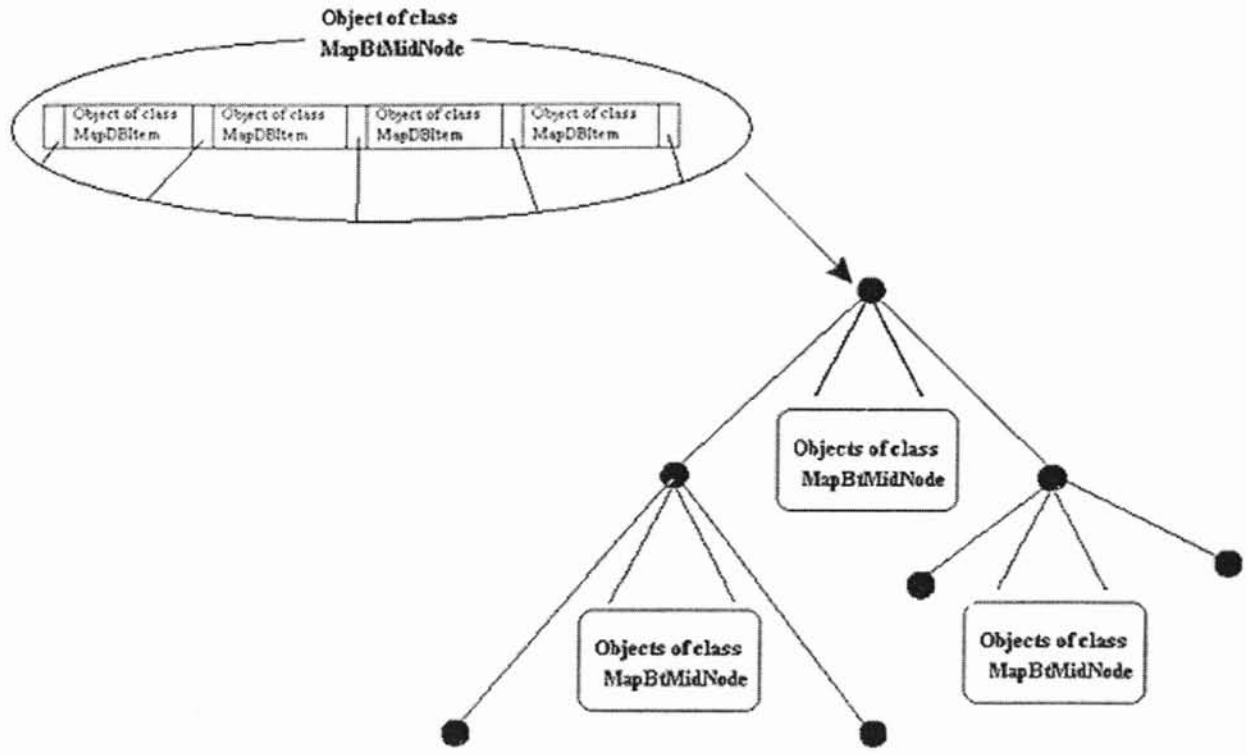


Figure 3.17 An Instance of class MapBTree

## CHAPTER IV

### PERFORMANCE ANALYSIS

This chapter describes the timing experiments and presents the test results. The tests include raw SQL performance measurement and scalability measurement under various computer systems. The purposes of the performance analysis for B-Tree object-relational access interface are:

1. To determine the significant performance benefits by comparing SQL queries between a relational database and a database using the B-Tree access interface;
2. To gain insights into the feasibility of the B-Tree access interface.

#### 4.1 Platform, database and applications for performance analysis

The platform used for the tests is a Windows 2000 Server running SQL server 2000.

The kernel hardware and database table configurations include:

- Processor Speed: PentiumIII 500 MHz, PentiumVI 1900 MHz (dual-processor)
- Memory Size: 512Mb, 2000Mb
- Test database: **FluidTestData**, a hydraulic engineering database that includes five tables:

Table Name: <b>FluidProperty</b> Primary key: FID; Foreign key: N/A		
Field Name	Data Type	Description
FID	Number	Fluid ID
Name	Text	Fluid Name
Manufacturer	Text	Fluid Manufacturer
ProductNo	Number	Fluid Manufacturer's Production No.
Use	Text	Usage of Fluid

Table 4.1 Definitions of Test Database Table FluidProperty

Table Name: <b>FlowData</b>		
Primary key: Time; Foreign key: FID		
<b>Field Name</b>	<b>Data Type</b>	<b>Description</b>
Index	Number	Data Index
Time	Text	Time of Fluid Flow
FValue	Double	Test Data Value of Fluid Flow
FID	Number	Fluid ID

Table 4.2 Definitions of Test Database Table FlowData

Table Name: <b>PressureData</b>		
Primary key: Time; Foreign key: FID		
<b>Field Name</b>	<b>Data Type</b>	<b>Description</b>
Index	Number	Data Index
Time	Text	Time of Fluid Flow
PValue	Double	Test Data Value of Fluid Pressure
FID	Number	Fluid ID

Table 4.3 Definitions of Test Database Table PressureData

Table Name: <b>TemperatureData</b>		
Primary key: Time; Foreign key: FID		
<b>Field Name</b>	<b>Data Type</b>	<b>Description</b>
Index	Number	Data Index
Time	Text	Time of Fluid Flow
TValue	Double	Test Data Value of Fluid Temperature
FID	Number	Fluid ID

Table 4.4 Definitions of Test Database Table TemperatureData

Table Name: <b>ViscosityData</b>		
Primary key: Time; Foreign key: FID		
<b>Field Name</b>	<b>Data Type</b>	<b>Description</b>
Index	Number	Data Index
Time	Text	Time of Fluid Flow
VValue	Double	Test Data Value of Fluid Viscosity
FID	Number	Fluid ID

Table 4.5 Definitions of Test Database Table ViscosityData

All of the above hardware and software are provided by BarDyne, Inc., Stillwater, Oklahoma. The test application is implemented under the Microsoft Visual C++ development platform version 6.0 on the Windows 2000 server. It is a windows dialog application that can build a B-tree object-oriented database access interface, open

database via ODBC, make SQL queries, display system information, query time, query results and evaluate query performance. Figure 4.1 and Figure 4.2 show sample output screens.

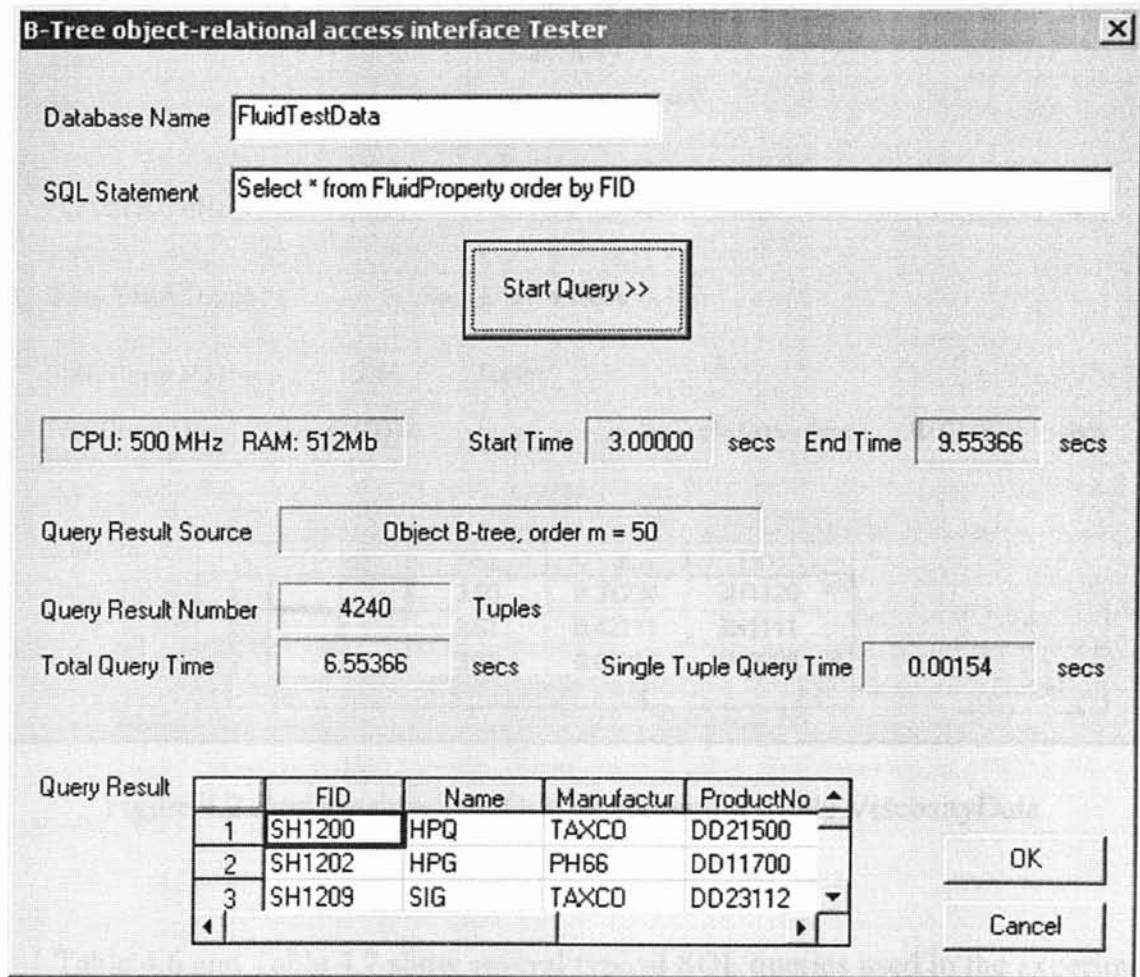


Figure 4.1 Performance Test Output Screen for table FluidPorperty

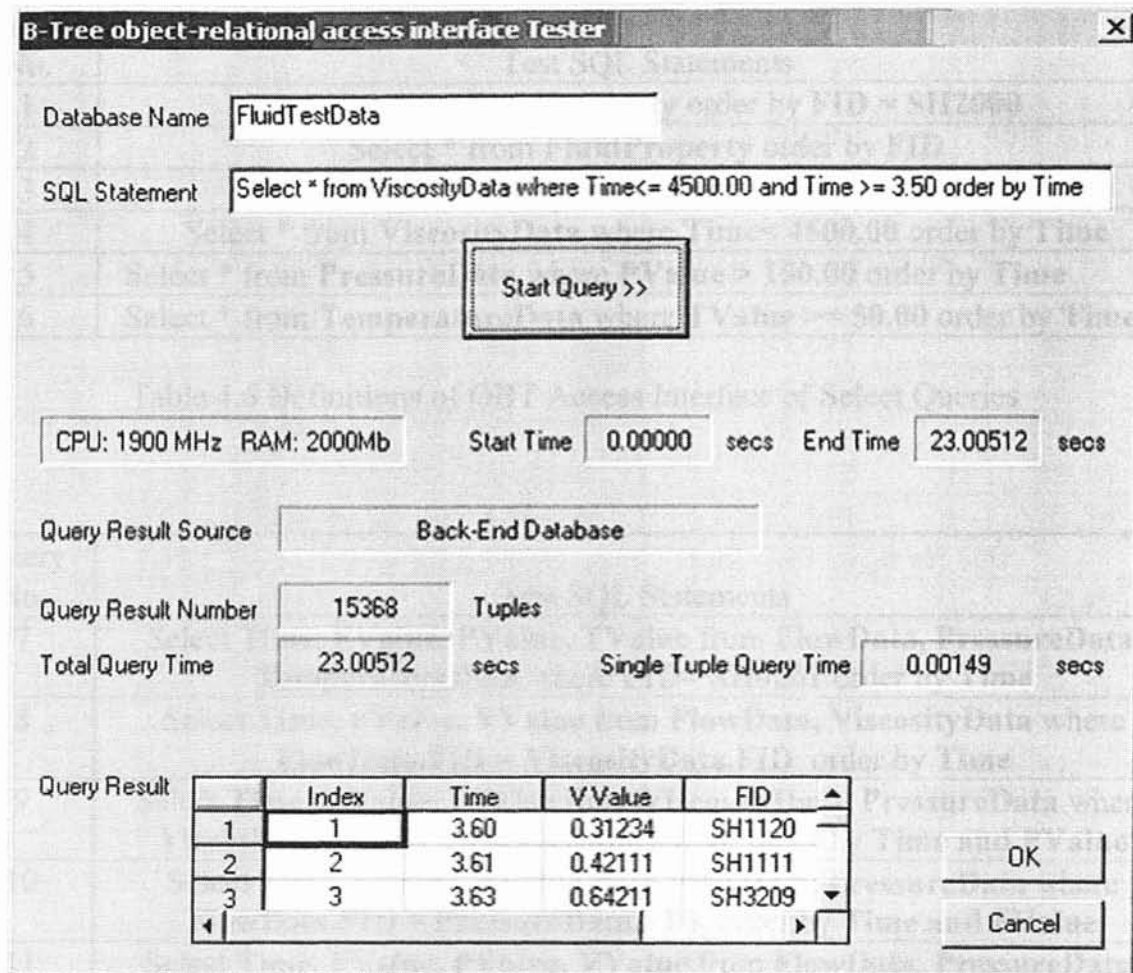


Figure 4.2 Performance Test Output Screen for Table ViscosityData

Table 4.6 and Table 4.7 show several typical SQL queries used in the experiment and represent the typical use of SQL queries, which involve selections and joins in the database. For every query, computer system configurations and B-tree order are given (table 4.8~ 4.11), and the size of the query result, the mean time required processing the query, and the percentage of time spent performing OBT search are listed in the table 4.8~ 4.11.

Query No.	Test SQL Statements
1	Select * from <b>FluidProperty</b> order by <b>FID = SH2000</b>
2	Select * from <b>FluidProperty</b> order by <b>FID</b>
3	Select * from <b>ViscosityData</b> order by <b>Time</b>
4	Select * from <b>ViscosityData</b> where <b>Time &lt; 4500.00</b> order by <b>Time</b>
5	Select * from <b>PressureData</b> where <b>PValue &gt; 150.00</b> order by <b>Time</b>
6	Select * from <b>TemperatureData</b> where <b>TValue &gt;= 50.00</b> order by <b>Time</b>

Table 4.6 Definitions of OBT Access Interface of Select Queries

Query No.	Test SQL Statements
7	Select <b>Time, FValue, PValue, TValue</b> from <b>FlowData, PressureData, TemperatureData</b> where <b>FID= SH0201</b> order by <b>Time</b>
8	Select <b>Time, FValue, VValue</b> from <b>FlowData, ViscosityData</b> where <b>FlowData.FID = ViscosityData.FID</b> order by <b>Time</b>
9	Select <b>Time, VValue, PValue</b> from <b>ViscosityData, PressureData</b> where <b>ViscosityData.FID = PressureData.FID</b> order by <b>Time and PValue</b>
10	Select <b>Time, FValue, PValue</b> from <b>FlowData, PressureData</b> where <b>FlowData.FID = PressureData.FID</b> order by <b>Time and FValue</b>
11	Select <b>Time, FValue, PValue, VValue</b> from <b>FlowData, PressureData, ViscosityData</b> where <b>FlowData.FID = PressureData.FID</b> and <b>ViscosityData.FID &gt; 100</b> order by <b>Time and VValue</b>

Table 4.7 Definitions of OBT Access Interface of Join Queries

## 4.2 Test Results

Query No.	Number of Query Result (tuples)	Database Query Time (s)	OBT Search Time (s)	% Time OBT Search in Database Query
CPU: Pentium 500 MHz; Memory: 512Mb; OBT order <b>m</b> : 50				
1	8360	70.335	66.572	94.65
2	25368	141.126	123.167	87.25
3	58000	293.413	240.230	81.87
4	92000	516.268	478.151	92.62
5	114585	790.232	710.257	89.88
6	150000	912.458	842.564	92.34
CPU: Pentium 500 MHz; Memory: 512Mb; OBT order <b>m</b> : 100				
1	8360	70.335	57.781	82.15
2	25368	141.126	110.509	78.31
3	58000	293.413	210.412	71.72
4	92000	516.268	434.967	84.25
5	114585	790.232	689.362	87.25
6	150000	912.458	812.810	89.09

Table 4.8 Performances of Select Query (A)

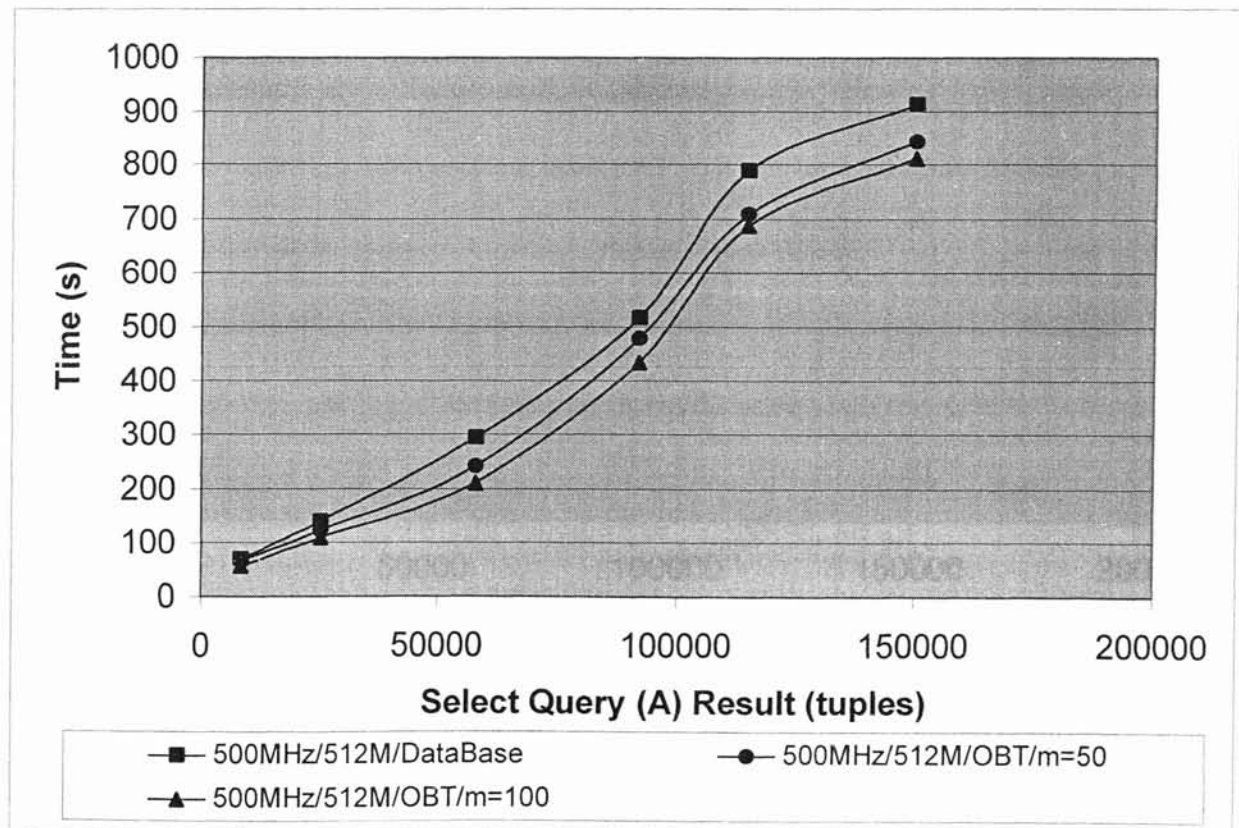


Figure 4.3 Performance Curve for Select Query (A)



Query No.	Number of Query Result (tuples)	Database Query Time (s)	OBT Search Time (s)	% Time OBT Search in Database Query
CPU: Pentium 1900 MHz; Memory: 2000Mb; OBT order <b>m</b> : 50				
1	8360	50.427	30.046	59.58
2	25368	103.156	57.283	55.53
3	58000	134.802	83.732	62.11
4	92000	189.381	123.309	65.11
5	114585	240.178	160.473	66.81
6	150000	371.168	220.618	59.44
CPU: Pentium 1900 MHz; Memory: 2000Mb; OBT order <b>m</b> : 100				
1	8360	50.427	8.312	16.48
2	25368	103.156	10.581	10.26
3	58000	134.802	17.609	13.06
4	92000	189.381	28.461	15.03
5	114585	240.178	47.523	19.79
6	150000	371.168	89.526	24.12

Table 4.9 Performances of Select Query (B)

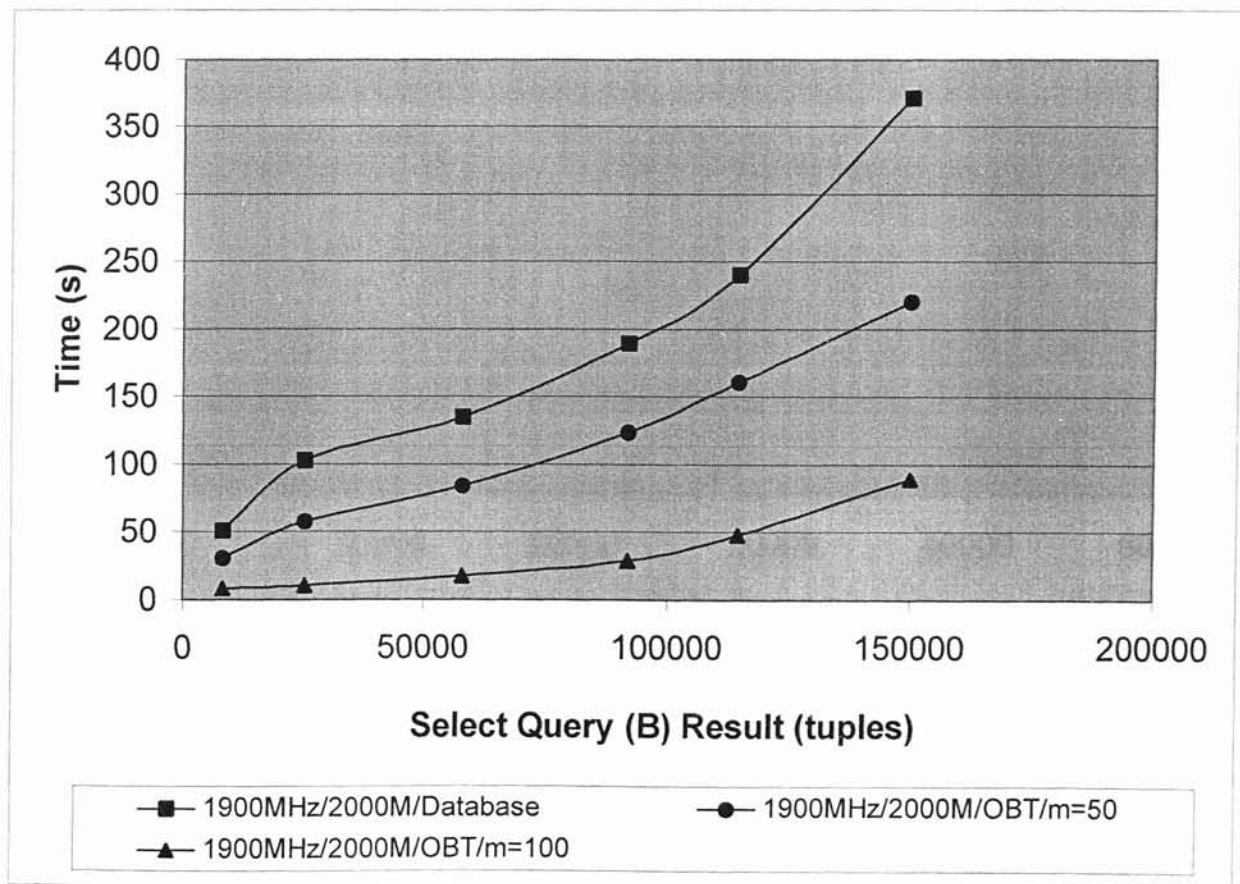


Figure 4.4 Performance Curve for Select Query (B)

Query No.	Number of Query Result (tuples)	Database Query Time (s)	OBT Search Time (s)	% Time OBT Search in Database Query
CPU: Pentium 500 MHz; Memory: 512Mb; OBT order <b>m</b> : 50				
7	3532	84.011	78.592	93.55
8	7074	109.347	89.709	82.04
9	15400	213.412	177.314	83.09
10	26900	288.167	268.581	93.20
11	45200	342.812	329.442	96.10
CPU: Pentium 500 MHz; Memory: 512Mb; OBT order <b>m</b> : 100				
7	3532	84.011	66.221	78.82
8	7074	109.347	81.701	74.72
9	15400	213.412	159.054	74.53
10	26900	288.167	259.319	89.99
11	45200	342.812	315.720	92.09

Table 4.10 Performances of Join Query (A)

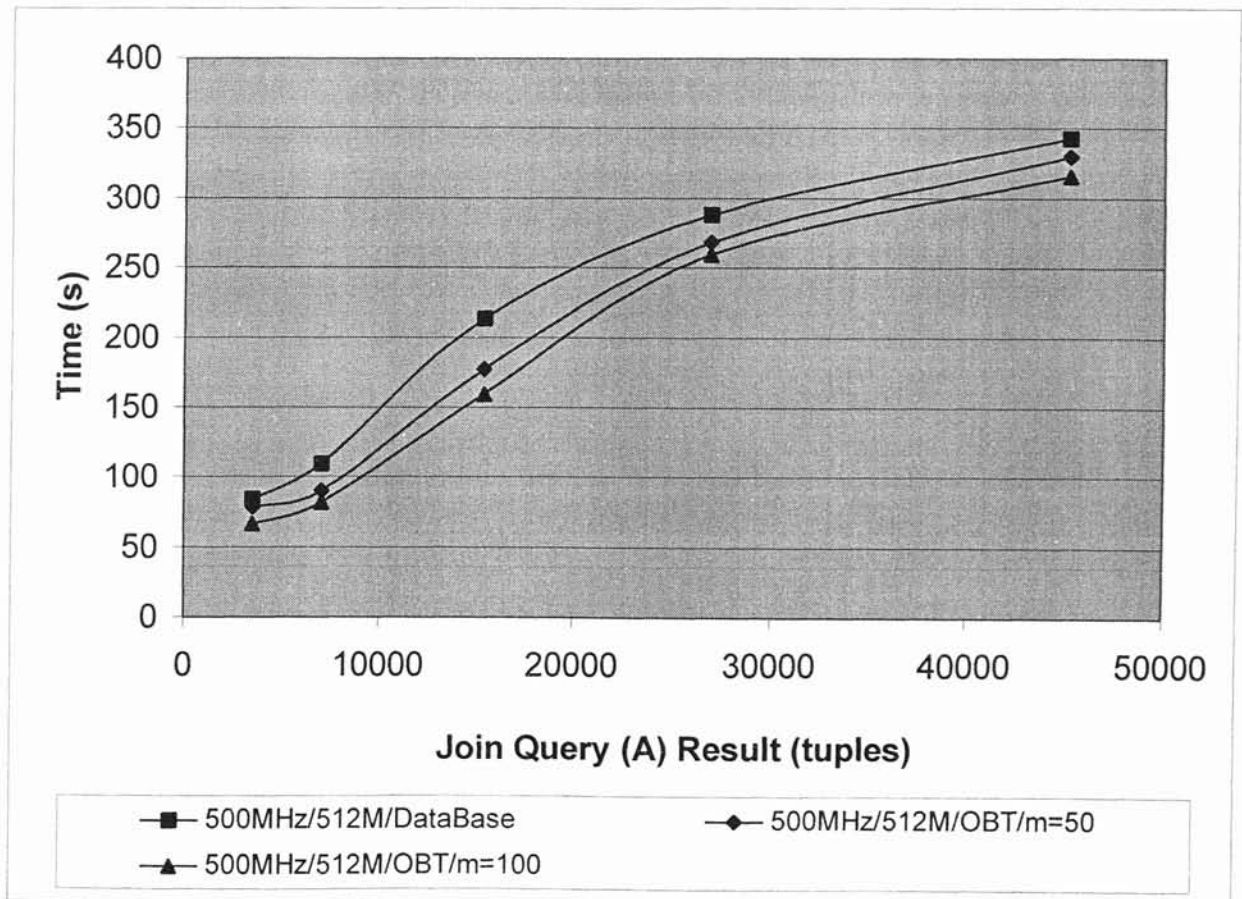


Figure 4.5 Performance Curve for Join Query (A)

Query No.	Number of Query Result (tuples)	Database Query Time (s)	OBT Search Time (s)	% Time OBT Search in Database Query
CPU: Pentium 1900 MHz; Memory: 2000Mb; OBT order <b>m</b> : 50				
7	3532	60.701	53.885	88.77
8	7074	71.312	62.471	87.60
9	15400	110.035	101.836	92.55
10	26900	133.107	122.230	91.83
11	45200	166.183	151.575	91.21
CPU: Pentium 1900 MHz; Memory: 2000Mb; OBT order <b>m</b> : 100				
7	3532	60.701	45.468	74.90
8	7074	71.312	52.426	73.52
9	15400	110.035	93.334	84.82
10	26900	133.107	114.319	85.89
11	45200	166.183	135.720	81.67

Table 4.11 Performances of Join Query (B)

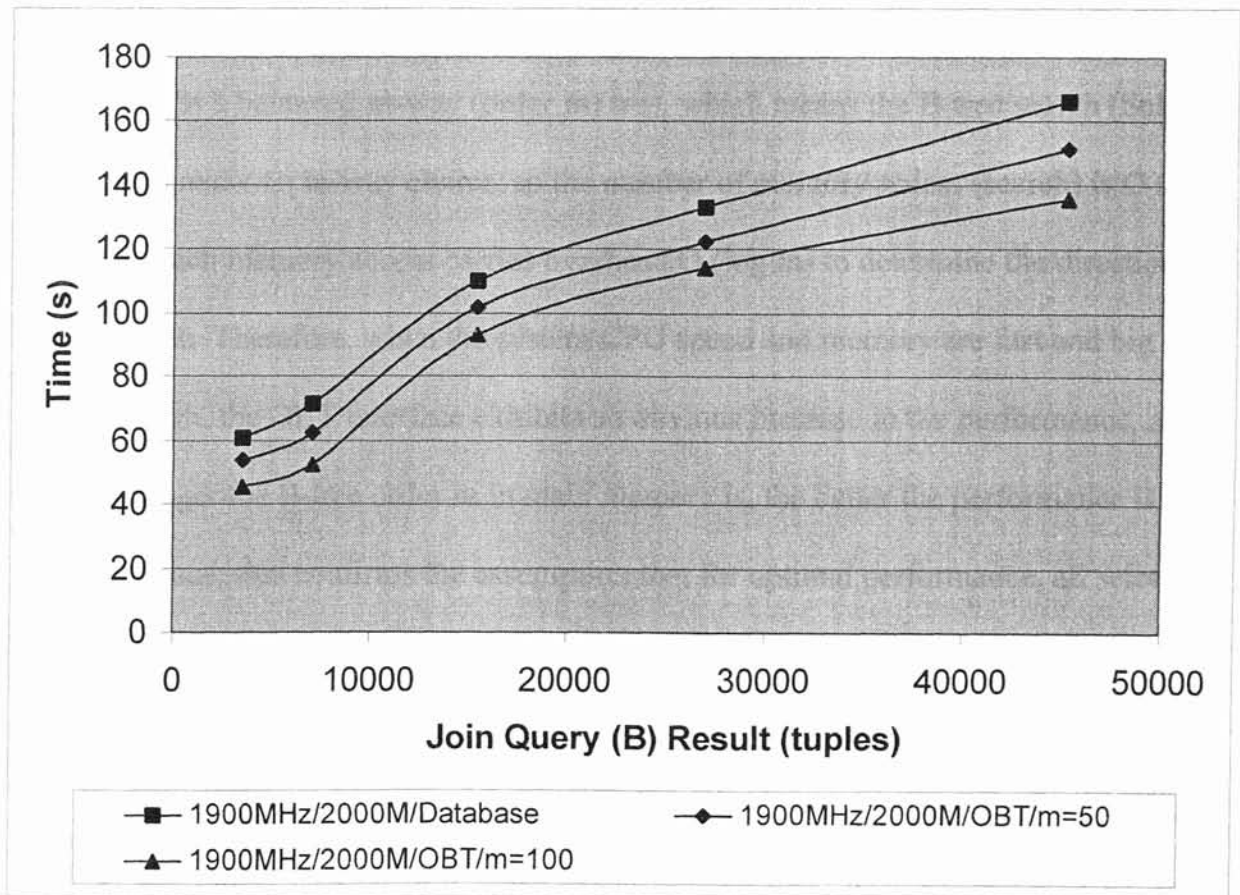


Figure 4.6 Performance Curve for Join Query (B)

### 4.3 Performance Test Summary

Table 4.8~4.11 present the performance test results obtained from select and join queries. Following results are observed:

- No matter how big the B-tree order  $m$  is, there is no obvious performance increase between database queries and OBT interface searches when system the CPU speed and memory are in low-end configurations. Because the system's main memory is too small to hold all object information, the operating system must page the most of objects to the hard drive, this causes the OBT interface search to happen on the hard drive instead of memory; it is as same as database query.
- OBT is a balanced  $m$ -way (order  $m$ ) tree, which means the B-tree search (Select) must make an  $m$ -way choice, so the number of memory access (search) is  $O(\log_m N)$ , each memory access carries overhead  $O(\log m)$  to determine the direction to branch. Therefore, when the system CPU speed and memory are fast and big enough, the OBT interface exhibits an obvious increase in the performance, and, the larger the B-tree order  $m$  in main memory is, the better the performance is obtained; this confirms the assumption that for optimal performance, all selection operations in the OBT interface should take place in main memory and B-tree needs to be relatively flat.
- The split operation in the OBT transforms a full node with  $2m - 1$  keys into two nodes with  $m - 1$  keys each and one key is moved into the parent node, the split algorithm will run in time  $O(m)$ . Queries 5 and 6 (Table 4.4) exhibit very poor join performance compared to Queries 1, 2, 3 and 4 when the OBT interface is

used. Join queries require a greater filtering during the search in order to retrieve objects from the memory. The OBT interface must treat the entire join relations as B-tree merge, split, and rotate operations. This significantly increases the system processing times as well as the number of attributes returned per tuple.

- Table 4.5 indicates that, if a primary key is used as an OBT search index for any relational table, on a high-end system configuration with a larger B-tree order number  $m$  (tree height is at most  $\lceil \log_{m/2} N \rceil$ ), the OBT interface yields satisfactory performance optimizations, since the running time of the search operation depends upon the height of the tree

## CHAPTER V

### CONCLUSIONS

The design and implementation of the object B-tree interface demonstrates how to achieve high performance in object-oriented applications linked to relational database.

The advantages and disadvantages of object B-tree interface are summarized below:

#### 5.1 Advantages vs. Disadvantages

##### Advantages:

1. High-speed object accessing abilities. All objects are memory-resident and organized as B-tree nodes. Using the primary key as the index of B-tree node key simplifies the object search minimizing the cost of evaluating the query;
2. Flexible data type processing ability. A B-tree class template can build complex data types according to the various relational table attributes;
3. The inheritance and encapsulation of the B-tree object provides a seamless link to the OODBMS and OO-applications;
4. All class templates and classes can be reusable.

Disadvantages:

1. A high-end hardware configuration is required; this increases the total system expense;
2. The two sets of SQL queries used for performance analysis indicate that performance of select queries is acceptable in the most cases, better than a direct database query. However, joins can depredate performance significantly in these situations because they are related to many-to-many relationships and not a native query for the OBT layer.

According to the performance analysis, the design of OBT layer must take into account the weaknesses and strengths of the relational data store. One-to-one relationships must be embedded into a single table using the primary key, but also using foreign keys to navigate one-to-many relationships while avoiding many-to-many relationships. We must keep the classes used to map tuples to node items as simple as possible except for cases that minimize database traffic. The multi-thread (a thread is a process running in the memory, multi-thread means several processes are running in main memory which are sharing one or more CPUs at the same time) is critical memory-resident object navigation and updates in order to maximize performance benefits for deploying large-scale object systems. The actual performance is dependent greatly upon how the application can take advantage of objects (data) stored in main memory. According to Loomis, "Persistence offers many of the performance benefits of an OODBMS while retaining the reliability and portability of the underlying relational database." [Loomis, 1994]

## 5.2 Research Contribution

The main research contribution in this thesis is the actual implementation of a flexible and multi-threaded main memory resident object B-tree interface for relational database access. This interface can be configured easily to satisfy different relational table structures and is in full working order. Unlike database systems that are almost universally based on a secondary storage (hard disk) architecture, the interface features a mapping from database tables to the memory resident B-tree architecture; this enables system to eliminate the information search and communication overhead, one of the major sources of poor performance, between the database and applications. Therefore, this kind of architecture design allows applications to operate directly on a persistent memory objects instead of tuples. This approach eliminates object moving between the database cache and persistent memory space, further improving the system performance.

## 5.3 Future Work

As shown by performance analysis, the good-performance of the object B-tree interface is based on the primary key index B-tree search; the foreign key search and the “join” operation are not good for the B-tree. One possible improvement is to reduce the number of B-tree operations: merge, split and rotation, based on the notion of a join index and rewriting the query to process an exhaustive B-tree search of the objects (tuples) from one B-tree (table) based on items (key values) obtained in the another B-tree (table).



This should result in improvements in the time required to the process depending upon the selectivity of the join index.

## BIBLIOGRAPHY

- [1] Cattell, R.G.G (1997). The Object Database Standard: ODMG (Revised Edition). New York, NY: Addison-Wesley Publishing Company, 1997
- [2] Codd, E. F. (1990). The Relational Model for Database Management (2<sup>nd</sup> Edition). New York, NY: Addison-Wesley Publishing Company, 1990.
- [3] Crotzer, A.D. (1975). Efficacy of B-Trees In An Information Storage And Retrieval Environment. Thesis. Oklahoma State University, 1975.
- [4] Chamberlin, Donald D., & Boyce, Raymond F. (1974). SEQUEL: A Structured English Query Language (Vol. 1). Ann Arbor, Michigan: SIGMOD Workshop, 1974.
- [5] Date, C. J. (1995). An Introduction to Database Systems. Reading, MA: Addison-Wesley Publishing Company, 1995.
- [6] Date, C. J., & Darwen, Hugh (1993). The SQL Standard, Third Edition. New York, NY: Addison-Wesley Publishing Company, 1993.
- [7] Fleming, C.C., & Halle, B.V. (1989). Handbook of relational database design. New York, NY: Addison-Wesley Publishing Company, Reading, 1989.
- [8] Garcia-Molina, H., Ullman, J., and Widom, J. (1998) Database System Implementation. Upper Saddle River, NJ: Prentice Hall, 1998.
- [9] Gray, J., and Reuter, A. (1993). Transaction Processing: Concepts and Techniques. San Francisco, CA: Morgan Kaufmann, 1993.

- [10] Halpin, Terry (1995). Conceptual Schema & Relational Database Design (2<sup>nd</sup> Edition). Englewood Cliffs, NJ: Prentice Hall, 1995.
- [11] Jordan, D (1996). C++ Object Databases: Programming with the ODMG Standard. New York, NY: Addison-Wesley Publishing Company, 1996.
- [12] Keller, A.M., Agarwal, S. and Jensen, R. (1993). Enabling The Integration of Object Applications With Relational Databases. Proceedings of 1997 ACM SIGMOD Conference, San Jose, CA, 1997.
- [13] Knuth, Donald (1973). The Art of Computer Programming. New York: Addison Wesley Publishing Company, Melno Park, California, 1973.
- [14] Loomis, M. (1994). Querying Object Database. Journal of Object-Oriented Programming, 7(3): 56-78. June, 1994.
- [15] Loomis, M. (1995). Object Databases: The Essentials. Reading, MA: Addison-Wesley Publishing Company, 1995.
- [16] Matheus, C. J., Chang, P. K., & Piatetsky-Shapiro. (1993). Systems for Knowledge Discovery in Databases. En IEEE Transactions on Knowledge and Data Engineering, vol. 5(6), December, 1993.
- [17] Shaffer, Clifford A. (2000) A Practical Introduction to Data Structures and Algorithm Analysis (C++ 2<sup>nd</sup> Edition). Englewood Cliffs, NJ: Prentice-Hall, 2000.
- [18] Simon, Richard J. (1996). Windows 95 multimedia & ODBC API Bible. Corte Madera, CA: Waite Group Press, 1996.

- [19] Ullman, J. (1988). Principles of Database and Knowledge-base Systems (Vol. II). Rockville, MD: Computer Science Press, 1988.
- [20] Ullman, J. and Widom, J (2001). A first course in Database Systems (2<sup>nd</sup> Edition). NY: Prentice Hall, 2001.
- [21] Ullman, J. (1991). A comparison between deductive and object-oriented database systems.  
DOOD Conference, Munich, Germany, 1991.
- [22] Weiss, M. A. (1997). Data Structures and Algorithm Analysis in C (2<sup>nd</sup> Edition), Menlo Park, CA: Addison-Wesley Publishing Company, 1997.
- [23] Wirth, Niklaus (1986). Algorithms + Data Structures = Programs. Englewood Cliffs, NJ: Prentice-Hall, 1986.
- [24] Zaratian, Beck (1998) Microsoft Visual C++ 6.0 Programmer's Guide. Redmond, WA: Microsoft Press, 1998

## GLOSSARY

**aggregation** A part-of relationship between a component object and an aggregate object.

The process of transforming data from a detailed to a summary level.

**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.

**constraint** A restriction placed on a form, table, field or other object to either force an action to happen in a certain way, or to prevent an action from happening all together. Constraints enforce referential integrity, and promote fault tolerance.

**DBMS** Database management systems.

**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 ois-ao or okind-of-o

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.

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

**OBT** Object B-Tree

**OODBMS** Objected-oriented database management system that can be used to store and retrieve objects.

**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.

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

**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.

**stored procedure** A program running in the database that can take complex actions based on the inputs you send it. Using a stored procedure is faster than doing the same work on a database client, because the program runs right inside the database server.

**table** The grouping of information in a relational database. Tables are composed of columns and rows.

**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.

**trigger** A program in a relational database that gets called each time a row in a table is inserted, changed, or deleted. Triggers allow you to check that any changes are correct before they are accepted.

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

## APPENDIX A

### CLASS SCHEMA DEFINITION FOR B-TREE OBJECT-RELATIONAL DATABASE ACCESS INTERFACE

```
class CMapDB
{
public:
    CMapDB();
    virtual ~CMapDB();

//variables
public:
    SQLRETURN m_hReturnCode;
    SQLHENV m_hEnv;
    SQLHDBC m_hODBC;
    HSTMT m_hDesc;

//operations
public:
    bool DBOpen( const char* strDSN, const char* strUserName="", const char*
strPWD="" );
    bool DBClose();
    bool SQLDBQuery( const char* strSQL );
    bool IsDBOpen() { return m_bIsOpen; }
    void ThrowDBError( SQLSMALLINT ErrorHandler=SQL_HANDLE_STMT );
    SQLRETURN GetDBReturnCode() { return m_hReturnCode; }
    bool DBTransactions();
    bool SQLDBCommit();
    bool SQLDBRollback();

private:
    string m_strDSN;
    string m_strUserName;
    string m_strPWD;
    string m_strQueryReturn;
    bool m_bIsOpen;
    bool m_BAutoCommit;
};
```

Figure 3.3 CMapDB Class Schema Definition



```

class CMapRecordset :
{
    // constructor/destructor
    public:
        CMapRecordset();
        CMapRecordset( CMapDB& db );
        virtual ~CMapRecordset();
        void ResetContent();
        void operator <<( const char* statement );
        bool ExecMapInsertSQL(const char* strSQL);
        bool ExecMapUpdateSQL(const char* strSQL);
        bool ExecMapDeleteSQL(const char* strSQL);
        bool ExecMapSelectSQL( const char* strSQL );
        bool ExecMapFetch();

        void SetMapDBOrderBy( const char * strOrderBy );
        void SetMapDBGroupBy( const char* strGroupBy );
        void SetMapDBWhere( const char* strWhere );

    public:
        const char* SQLGetValue( int nRow, int nDataType);
        int GetColumnDataType( int nColumnNumber );
        int GetColumnNumber();
        void SetHDBC( HDBC* hdbc );
        bool MapSQLAllocStmt();
        void MapSQLFreeStmt( SQLUSMALLINT a_uType=SQL_DROP );

    protected:
        void ThrowError();

    private:
        SQLRETURN m_hReturnCode;
        SQLHDBC m_hODBC;
        SQLHSTMT m_hStatement;

};

```

Figure 3.4 CMapRecordset Class Schema Definition

```

class CMapException
{
public:
    CMapException() :m_nRetCode( 0 ) {};
    CMapException( SQLRETURN return ) { m_nRetCode = return; }
    ~CMapException() {};

    SQLRETURN m_nRetCode;
    string m_strError;
    string m_strStatement;
    bool m_bExceptionAgain;
    void Delete() { delete this; }

private:
    CMapException( const CMapException& o ) {};

};

```

Figure 3.5 CMapException Class Schema Definition

```

class MapBtItemType
{
public:
    MapBtItemType();
    ~MapBtItemType();

    int m_nIndex; // the index of this item in the host node

    virtual void OnSetStatus(); //item status: update/delete/active
    virtual int OnGetStatus(); //item status: update/delete/active
    virtual void OnSetValue(); //setup item values
    virtual void OnGetValue(); //return item values
    virtual void OnPrintOutValue(); //print out item values

private:
    int m_nStatus; // S_UPDATE/S_DELETE/S_ACTIVE

};

```

Figure 3.8 MapBtItemType Class Schema Definition

```

template <class KeyType, class ValueType >
class MapDBItem : public MapBtItemType
{
friend class MapBTree<KeyType, ValueType>;
friend class MapBtMidNode<KeyType, ValueType>;

public:

    MapDBItem(const KeyType& key, const ValueType&
value):m_tPrimaryKey(key), m_tValue(value){}
    MapDBItem(const KeyType& key): m_tPrimaryKey(key) {};
    virtual ~MapDBItem(){}

public:
    int m_nField;
    string m_strTableName;
    KeyType m_tPrimaryKey;
    ValueType m_tValue;

    virtual int operator<(const MapBtItemType& Operand) const
    {
        return (m_tPrimaryKey < ((const MapDBItem<KeyType,
ValueType>&)Operand).m_tPrimaryKey);
    }

    virtual int operator==(const MapBtItemType& Operand) const
    {
        return (m_tPrimaryKey == ((const MapDBItem<KeyType,
ValueType>&)Operand).m_tPrimaryKey);
    }

    virtual int operator>(const MapBtItemType& Operand) const
    {
        return (m_tPrimaryKey > ((const MapDBItem<KeyType,
ValueType>&)Operand).m_tPrimaryKey);
    }

    virtual const MapBtItemType& operator=(const MapBtItemType& Operand)
    {
        m_tPrimaryKey=((const MapDBItem<KeyType,
ValueType>&)Operand).m_tPrimaryKey;
        m_tValue=((const MapDBItem<KeyType,
ValueType>&)Operand).m_tValue;
        return *this;
    }
}

```

```

    MapBTree<KeyType, ValueType>* m_pBTree; // the parent B-Tree
    MapBtMidNode<KeyType, ValueType> *m_pHostNode; // the node hosts this
item
    MapBtMidNode<KeyType, ValueType> *m_pLeftNode; // left sub-tree of this
item
    MapBtMidNode<KeyType, ValueType> *m_pRightNode; // right sub-tree of
this item

    virtual MapBtMidNode<KeyType, ValueType> *OnGetNode(int nNode); //0: host; 1:
left; 2: right

    virtual void OnSetStatus(); //item status: update/delete/active
    virtual int OnGetStatus(); //item status: update/delete/active
    virtual void OnSetValue(CMapRecordset* pSet); //setup item values
    virtual void OnGetValue(); //return item values
    virtual void OnPrintOutValue(); //print out item values

};

```

Figure 3.9 Schema Definition of Template Class MapDBItem

```

template <class KeyType, class ValueType >
class MapBtNodeType
{

    friend class MapBTree<KeyType, ValueType>;
    friend class MapBtMidNode<KeyType, ValueType>;

public:

    MapBtNodeType(MapBtMidNode<KeyType, ValueType > *pParent, MapBTree
*pBTree = NULL);
    virtual ~MapBtNodeType();

    int m_nNodeIndex;
    int m_nItemCount;           // Number of Items stored in the current node
    bool m_bIfLeaf;
    MapBtItemType m_Item[MaxItems]; // indexing starts at 0, Items in the node
    MapBtMidNode<KeyType, ValueType >* m_ChildNode[MaxItemsPlusOne]; //
pointers to child nodes

public:
    bool IsNodeFull();

    MapBTree* GetHostBTree() const {return m_pBTree;}
    MapBtMidNode<KeyType, ValueType >* GetParentNode() const {return
m_pParentNode;}
    virtual MapBtItemType* operator[](int nIndex) const = NULL;
    virtual MapBtMidNode<KeyType, ValueType >* GetChildNode(int
nChildIndex) const
        {return m_ChildNode[nChildIndex];}

protected:

    MapBTree* m_pBTree;
    MapBtMidNode<KeyType, ValueType >* m_pParentNode;

};

```

Figure 3.10 Schema Definition of Template Class MapBtNodeType

```

template <class KeyType, class ValueType >
class MapBtMidNode : public MapBtNodeType
{
public:
    MapBtMidNode(MapBtMidNode<KeyType, ValueType > *pParent, MapBTree
*pBTree = NULL);
    MapBtMidNode(MapBtMidNode<KeyType, ValueType > *pParent, MapBTree
*pBTree, MapBtNodeType *pOldRoot);
    ~MapBtMidNode();

    virtual void AddItem(const MapDBItem<KeyType, ValueType> objMapBtItem,
int nItemIndex);
    void AddItem(MapDBItem<KeyType, ValueType>& objMapBtItem, int
nItemIndex);
    void AddItemAt(MapDBItem<KeyType, ValueType> objMapBtItem, int
nItemIndexAt, MapBtNodeType *pNode);

    virtual void DeleteItem(int nItemIndex);
    virtual bool SearchItem(const MapDBItem<KeyType, ValueType> ObjTarget, int
& nLocation) const;

    virtual MapDBItem<KeyType, ValueType>* operator[](int nIndex) const =
NULL;
    virtual MapBtMidNode<KeyType, ValueType >* GetChildNode(int
nChildIndex) const {return m_ChildNode[nChildIndex];}

    int GetItemIndex(MapDBItem<KeyType, ValueType> objMapBtItem) const;
    int GetItemIndex(MapDBItem<KeyType, ValueType>* pMapBtItem) const;

    void SetItem(int nIndex, MapDBItem<KeyType, ValueType>& objItem)
        { m_Item[nIndex] = objItem; objItem.m_pHostNode = this; }

    void SetChildNode(int nIndex, MapBtMidNode<KeyType, ValueType >
*pChild)
        { m_ChildNode[nIndex] = pChild; }

    bool IsNodeFull();
    bool IsNodeFull(MapBtNodeType *pNode);
};

```

Figure 3.11 Schema Definition of Template Class MapBtMidNode

```

template <class KeyType, class ValueType >
class MapBTree
{
    friend class MapBTreeIterator<KeyType, ValueType>;

protected:
    MapBtMidNode<KeyType, ValueType> m_tHd;    // MapBTree header
    MapBtMidNode<KeyType, ValueType>* m_tNodeCache; // Node cache
    MapBtMidNode<KeyType, ValueType>* m_pRoot; // Pointer to the root node

public:
    string m_strTableName;
    KeyType m_kPrimaryKey;
    long m_nNumNodes; // number of nodes in the B-tree
    int m_nNodeSize; // number of bytes per node
    int m_nOrder;    // order of the B-tree

    MapBtMidNode<KeyType, ValueType> m_tCurrentNode; // storage for current
node being worked on

    bool InsertItem(MapDBItem<KeyType, ValueType> &tItem);
    bool UpdateItem(MapDBItem<KeyType, ValueType> &tItem, int nNodeIndex,
int nItemIndex);
    bool DeleteItem(MapDBItem<KeyType, ValueType> &tItem);
    bool DeleteItem(int nNodeIndex, int nItemIndex);

    bool InsertNode(MapBtMidNode<KeyType, ValueType> &tNode);
    bool DeleteNode(MapBtMidNode<KeyType, ValueType> &tNode);
    bool DeleteNode(int nNodeIndex);

    void LeftBalance (MapBtMidNode<KeyType, ValueType> *pLNode, int
nIndex);
    void RightBalance (MapBtMidNode<KeyType, ValueType> *pRNode, int
nIndex);
    void RestoreBalance(MapBtMidNode<KeyType, ValueType> &tNode, int
nIndex);
    void BalanceNode(MapBtMidNode<KeyType, ValueType> *pNode, int nIndex);

    void Split();
    void SplitNode(MapBtMidNode<KeyType, ValueType> *pNode, int nIndex);
    void RotateRight(MapBtMidNode<KeyType, ValueType> &tNode, int nIndex);
    void RotateLeft(MapBtMidNode<KeyType, ValueType> &tNode, int nIndex);

```

```

        void MoveToLeft(MapBtMidNode<KeyType, ValueType> *pLeftSibNod, int
nParentPos);
        void MoveToRight(MapBtMidNode<KeyType, ValueType> *pRightSibNod, int
nParentPos);

        void RightMerge (MapBtMidNode<KeyType, ValueType> *pRNode, int
nIndex);
        void Merge(MapBtMidNode<KeyType, ValueType> &tNode, int nIndex);

        void AppendFrom(MapBtMidNode<KeyType, ValueType> *pSrcNod, int
nStartPos, int nStopPos);
        void Append(MapBtMidNode<KeyType, ValueType> *pObjNod,
MapBtNodeType& Node);
        void Append(CMapDBItem<KeyType, ValueType> &itm);

        static void PrintNode(MapBtMidNode<KeyType, ValueType> tNode);
        static void PrintTree(MapBtMidNode<KeyType, ValueType> tTree, int
nStartPos);

public:

        MapBTree();
        MapBTree(const MapBTree<KeyType, ValueType>& objMapBTree);
        ~MapBTree();

};

```

Figure 3.12 Schema Definition of the template class MapBTree



```

template <class KeyType, class ValueType >
class MapBTreeIterator
{
private:
    const MapBTree<KeyType,ValueType>* m_pBTree;    //btree being iterated
    int m_nCursor; //current position in btree
    bool m_bDirection; //iteration direction
    MapBTreeIterator() : m_pBTree(0), m_nCursor(0) { }

public:
    MapBTreeIterator(const MapBTree<KeyType,ValueType> *BTree, bool
bPathDir);
    MapBTreeIterator(const MapBTreeIterator<KeyType,ValueType> &Iter);
    ~MapBTreeIterator() { }
    MapBTreeIterator &operator=(const MapBTreeIterator<KeyType,ValueType>
&rhs);

};

```

Figure 3.13 Schema Definition of the class template MapBTreeIterator

```

class CMapParser
{
public:
    CMapParser() :m_nRetCode( 0 ) {}
    ~CMapParser();

//attributions
public:
    string m_strError;
    bool m_bExceptionAgain;

    CMapDB m_DB;
    CMapRecordset *m_pSet;

//operations

    BOOL OnOpenDB();
    BOOL OnCloseDB();
    void OnParserSQL();
    void OnSendRawSQL();
    void OnSearchBTree();

    void OnSetSQL(string strSQL);
    string OnGetSQL(string strSQL);

private:

    void OnBuldOBT();
    void OnDeleteOBT();
    string strTableName[50];
    string strFielsName[50][20];
    string m_strSQLStatement;
};

```

Figure 3.14 Schema Definition of CMapParser



Qing Chen

Candidate for the Degree of

Master of Science

Thesis: AN INTERFACE BETWEEN OBJECT-ORIENTED DATABASES AND  
RELATIONAL DATABASES USING B-TREE OBJECT-RELATIONAL ACCESS  
INTERFACE

Major Field: Computer Science

Biographical:

Personal Data: Born in Changchun, Jilin, P. R. China, Oct 10, 1971, the only son  
of Mr. Chaoda Chen and Mrs. Zaiying Hu

Education: Graduated from the No. 84 High School of Xi'an, Shaanxi, P. R.  
China, in July 1989; received the Bachelor of Science from Xi'an Jiaotong  
University, Xi'an, China, in July 1994; Completed the requirements for  
Master of Science at Oklahoma State University in December, 2002.

Professional Experience: Employed by Xi'an Petroleum Institute, Xi'an, Shaanxi,  
China, as a Teacher, August 1994 to July 1997; Employed by Vysys Inc.  
(National Inter Bank), Tulsa, Oklahoma, as a Senior System Analyst,  
January 2000 to January 2002. Employed by Businet LLC, Tulsa,  
Oklahoma, as a Senior Design Engineer from August 2002.